NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

DESIGN AND EVALUATION FOR THE END-TO-END DETECTION OF TCP/IP HEADER MANIPULATION

by

Ryan M. Craven

June 2014

Dissertation Supervisor: Robert Beverly

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	06-20-2014	Dissertation 09-27-2010 to 06-20-2014

4. TITLE AND SUBTITLE

DESIGN AND EVALUATION FOR THE END-TO-END DETECTION OF TCP/IP HEADER MANIPULATION

5. FUNDING NUMBERS

NSF: CNS-1213155
SPAWAR: 2013-TIKI-030

6. AUTHOR(S)

Ryan M. Craven

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Science Foundation
4201 Wilson Blvd.
Arlington, VA 22230

SPAWARSYSCEN Atlantic
PO Box 190022
North Charleston, SC 29419

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Understanding, measuring, and debugging IP networks, particularly across administrative domains, is challenging. One aspect of the challenge are transparent middleboxes, which are now common in today's Internet. In-path middleboxes that modify packet headers are typically transparent to a TCP, yet can impact the end-to-end performance of its connections. Of equal importance, middleboxes cause architectural ossification that hinders network protocol evolution—new options or redefined header fields are often misconstrued, modified, or disabled. We develop TCP HICCUPS to reveal packet header manipulation to both endpoints of a TCP connection. HICCUPS adds a lightweight tamper-evident seal to TCP that is incrementally deployable and introduces no new options. HICCUPS provides an optional feature, AppSalt, that allows applications to request added protection for their connection's integrity, making it more difficult for middleboxes to falsify integrity values. HICCUPS is implemented in both an operating system patch to the Linux TCP stack as well as a set of cross-platform user-space tools. To evaluate HICCUPS, we deploy it to a diverse set of Internet nodes spread across 197 networks and 48 countries, measuring packet header manipulations on over 26 thousand directed port/path pairs. We discover over 11 thousand instances of unique non-NAT in-path packet header modifications across those flows, all with the potential to negatively affect TCP performance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer networks, TCP/IP, Internet measurement, middleboxes, packet header modifications, HICCUPS	189
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DESIGN AND EVALUATION FOR THE END-TO-END DETECTION OF TCP/IP HEADER MANIPULATION

Ryan M. Craven
Civilian, Space and Naval Warfare Systems Center Atlantic
B.S., Clemson University, 2007
M.S., Clemson University, 2010

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Ryan M. Craven

Approved by: Robert Beverly
Assistant Professor of Computer Science
Dissertation Supervisor and Committee Chair

Geoffrey Xie
Professor of Computer Science

Mark Gondree
Research Assistant Professor of Computer Science

Preetha Thulasiraman
Assistant Professor of Electrical Engineering

Steven Bauer
Research Affiliate, MIT

Approved by: Peter Denning
Chair, Department of Computer Science

Approved by: O. Douglas Moses
Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Understanding, measuring, and debugging IP networks, particularly across administrative domains, is challenging. One aspect of the challenge are transparent middleboxes, which are now common in today's Internet. In-path middleboxes that modify packet headers are typically transparent to a TCP, yet can impact the end-to-end performance of its connections. Of equal importance, middleboxes cause architectural ossification that hinders network protocol evolution—new options or redefined header fields are often misconstrued, modified, or disabled. We develop TCP HICCUPS to reveal packet header manipulation to both endpoints of a TCP connection. HICCUPS adds a lightweight tamper-evident seal to TCP that is incrementally deployable and introduces no new options. HICCUPS provides an optional feature, AppSalt, that allows applications to request added protection for their connection's integrity, making it more difficult for middleboxes to falsify integrity values. HICCUPS is implemented in both an operating system patch to the Linux TCP stack as well as a set of cross-platform user-space tools. To evaluate HICCUPS, we deploy it to a diverse set of Internet nodes spread across 197 networks and 48 countries, measuring packet header manipulations on over 26 thousand directed port/path pairs. We discover over 11 thousand instances of unique non-NAT in-path packet header modifications across those flows, all with the potential to negatively affect TCP performance.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

List of Acronyms and Abbreviations

3WHS three-way handshake

ACK acknowledgment

AH Authentication Header

API application programming interface

AS autonomous system

BGP Border Gateway Protocol

BTNS Better-than-nothing-security

CAIDA Cooperative Association for Internet Data Analysis

CDN content delivery network

CE congestion encountered

CRC cyclic redundancy check

CWR congestion window reduced

DF don't fragment

DoS denial of service

DSCP differentiated services code point

ECE ECN echo

ECN Explicit Congestion Notification

ECT ECN-capable transport

EFF Electronic Frontier Foundation

ESP Encapsulating Security Payload

FEC forward error correction

FIN finish

HICCUPS Handshake-based Integrity Check of Critical Underlying Protocol Semantics

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

ICSI International Computer Science Institute

ICV integrity check value

IDS intrusion detection system

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IHL Internet header length

IP Internet Protocol

IPID IP identification

IPsec Internet Protocol Security

ISN initial sequence number

ISP internet service provider

MAC message authentication code

MITM man-in-the-middle

MSS maximum segment size

NAT network address translation

NS nonce sum

OSI	Open Systems Interconnection
PKI	public key infrastructure
PMTUD	path maximum transmission unit discovery
RFC	request for comments
RST	reset
RTT	round trip time
SACK	selective acknowledgment
SDN	software-defined networking
SIMPLE	Software-defIned Middlebox PoLicy Enforcement
SSL	Secure Sockets Layer
SYN	synchronize
TCP	Transmission Control Protocol
TCP-AO	TCP Authentication Option
TFO	TCP Fast Open
TLS	Transport Layer Security
ToS	type of service
TTL	time-to-live
UDP	User Datagram Protocol
URL	uniform resource locator
VPN	virtual private network
WAN	wide area network
XOR	exclusive or

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Picture a scenario where two endpoints, Alice and Bob, communicate across the Internet. Traffic sent between Alice and Bob transit a variety of links, as well as through a diverse array of switches, routers, and intermediate network devices known as middleboxes. Alice's TCP stack is responsible for utilizing this path to accomplish the reliable reassembly of her data stream at Bob's end of the connection in as an efficient manner as possible. Prior to the connection, Alice's TCP knows absolutely nothing about the conditions of the path to which her data will transit. In other words, Alice's TCP has no idea *a priori* how to properly answer many critical decisions that must be made over the course of every connection:

- How fast can I send?
- How many segments should I send at once?
- Did Bob receive my data intact?
- Was a piece missing?
- Was the data stream in the right order?
- Was it free of transmission errors?

Various techniques have been integrated into TCP so that as the connection progresses, Alice's TCP can **infer** the state of the path and then apply a strategy to best answer the above questions. Instrumentation such as congestion control, sequence numbers, duplicate acknowledgments, selective acknowledgments, and checksums all provide actionable end-to-end path information that contributes to the robustness and effective performance of TCP on the modern (and often messy) Internet.

In this dissertation, we posit that TCP must provide an answer to another critical question about the transit path in order to maximize performance: **"Am I being misinterpreted?"**

As middleboxes become increasingly commonplace throughout the Internet, more and more data plane intelligence is added to the network resulting in a less flexible environment that is subject to behaviors that ossify packet header semantics and options fields designed for extensibility. Such behaviors make it difficult for protocol designers to innovate within TCP. Furthermore, even small occurrences of misconfigurations, non-standard implementations, and legacy deployments can negatively impact protocol interactions. In

addition to the directly resulting cases of degraded performance and connectivity issues, the impact of even a small number of cases across the Internet deters large providers from deploying new extensions designed to increase security and performance.

Such interactions are an under-appreciated issue that can have a widespread impact on the Internet and the evolutionary trajectories of the protocols that define it. For instance, we examine several real-world examples in this dissertation:

- **ISN translation and SACK**: A "security-enhancing" feature of a firewall at our institution was randomizing TCP initial sequence numbers, but not updating the corresponding selective acknowledgment blocks. The out-of-window SACK blocks confused TCP, resulting in poor performance.
- **ECN**: A network switch was assuming older semantics for the second byte of the IP header, overwriting ECN congestion status. The loss of feedback could result in congestion being inferred when there is none.
- **Window Scaling**: A middlebox operating on ports 80 and 443 was adding a window scaling option to SYNs that did not contain one, causing the remote TCP to assume the flow control was much smaller than it really was.
- **Multipath TCP**: A number of middleboxes stripped the initial MPCAPABLE option, inhibiting deployment of Multipath TCP.

Currently, TCP lacks the necessary instrumentation to ensure that it is being properly interpreted by the remote endpoint (i.e., that its packet headers have not been altered while in transit). The current state-of-the-art is an array of underused methods—all with some combination of deployment, incentive, or consistency issues that preclude integration into TCP. In contrast, our solution provides the methodology and tools for an automated and generally usable platform within TCP to expose such changes to packet header fields. The TCP Handshake-based Integrity Check of Critical Underlying Protocol Semantics (TCP HICCUPS) would allow TCP to infer in-path alteration of packet header fields, contributing to the body of end-to-end path information at its disposal and helping to answer the question of correct interpretation by the remote TCP.

HICCUPS is cooperative with currently deployed middleboxes, applying a tamper-evident seal to packet headers that allows devices to continue modifying packet headers as desired,

but revealing that modification to a connection's endpoints. HICCUPS is incrementally deployable and introduces no new options. HICCUPS benefits TCP by making it possible to reason about the correctness of how a path handles a specific protocol extension, letting TCP make more informed decisions about when it is safe to enable a new protocol extension, or disable one that is disrupted by a middlebox along a path.

HICCUPS also benefits users and network operators by enabling a new path diagnostic that can be generally used with any remote TCP that is HICCUPS-enabled. Such a diagnostic would not require any prior setup or cooperation and could be used in a manner similar to how `ping` and `traceroute` are used in unpremeditated debugging today.

The operation of HICCUPS is summarized by the following sequence of events:

1. The TCP initiating the active open computes an integrity value over a selection of header fields in its SYN packet using a publicly-defined integrity function.
2. The TCP *overloads* three fields within the IP and TCP headers:
   - the initial TCP sequence number
   - the initial IP identification
   - the initial TCP receive window size

   with the computed integrity value and a code to denote which fields were included in the integrity calculation.
3. The HICCUPS-enabled SYN is sent to a remote TCP.
4. A remote HICCUPS-enabled TCP recomputes the integrity calculation over the received SYN, using the set of fields specified by the sender.
5. The remote TCP compares the recomputed integrity with the integrity received in the three overloaded header fields. If two or more fields match, then it is inferred that the SYN's headers were unmodified.
6. After constructing the SYN-ACK, the remote TCP overloads the same three fields with an integrity check of the SYN-ACK, as well as a status code indicating whether the SYN's integrity matched.
7. The HICCUPS-enabled SYN-ACK is sent to the initiating TCP.
8. The initiating TCP recomputes integrity over the SYN-ACK and compares the values, returning the bi-directional header integrity status to the user.

At a high level, HICCUPS is specifically designed to be cooperative with middleboxes in order to minimize path traversal issues. We also readily allow middleboxes to continue their behavior modifying packet headers as they wish, but now inform each TCP of those changes. Our hope is that, by giving middleboxes no specific reason to disrupt HICCUPS, they will not try to alter its integrity values. However, middleboxes have a history of enforcing overly conservative security policies that block simple diagnostic traffic (e.g., ICMP), so we must take a realistic approach. We develop AppSalt as an additional layer of protection for the integrity values sent by HICCUPS, making it more difficult for a middlebox to introduce packet header modifications and then falsify the integrity values to hide the modification.

HICCUPS is implemented in both an operating system patch to the Linux TCP stack as well as a suite of cross-platform user-space tools. To evaluate HICCUPS, we deploy it to a diverse set of Internet nodes spread across 197 networks and 48 countries, measuring packet header manipulations over 26 thousand directed port/path pairs. We discover over 11 thousand instances of unique non-NAT in-path packet header modifications across those flows, all with the potential to negatively affect TCP performance. Of particular notability, we discover and analyze a new instance of window scaling detected on a real Internet path. Using results from HICCUPS, we are able to ignore window scaling on that path and find that bulk transfer performance more than doubles.

Acknowledgements

I would like to acknowledge and thank everyone that helped me along in my path to completing this dissertation. For everyone that contributed to this work, helped me solve a problem in my research, served as a sounding board for my ideas, or even just shared in mutual camaraderie with me over a beer at the Trident Room, I am profoundly grateful and sincerely appreciate your support.

First and foremost, I would like to thank my advisor, Dr. Robert Beverly. These last three years have been a challenging yet deeply rewarding experience and I can undoubtedly see that not only have I greatly improved as a researcher, but that your guidance and teaching have always been in my best interest. When I look back I notice tangible improvements in reading papers, writing, explaining my work and the works of others, and formulating good research ideas. All in all, I accomplished more with this work and gained more from the experience than I had ever thought I could—the sign of having had a truly great mentor.

Professionally, a great deal of gratitude is owed to all of the members of my dissertation committee: Drs. Geoffrey Xie, Mark Gondree, Preetha Thulasiraman, and Steve Bauer. Every one of you has taken time away from your own work to help me and I am fortunate to have such a great committee. I would additionally like to thank Steve for allocating us a measurement server and assisting with the PlanetLab measurements.

Many thanks to kc claffy and Young Hyun of CAIDA for helping us run measurements on Ark and offering their comments and support for our research idea. The additional dataset afforded by the Ark measurements greatly enriched the breadth of results and afforded us another level of insight to middlebox modifications.

I would also like to give a tip of the cap to Mark Allman and Nick Weaver at ICSI, and Justin Rohrer at NPS. The input from their experience and clever ideas not only strengthened this work it helped form parts of the basis for what made it a success.

My personal thanks go out to all of the other Ph.D. students during my time here at NPS: Jeff, James, Mike B., Mike C., Andrew, Brian, Alan, Donna, and Travis. Specifically, all the study sessions preparing for quals, the support through all the hurdles, that first trip to

Tahoe, the BBQs, and all the other words of advice and sanity along the way. To Ricky Gaylard and Bruce Carter at SPAWAR, Drs. Brooks and Berg at Clemson, and all of my friends and family, I am truly appreciative of all the encouragement you gave me to pursue my doctorate and to get where I am today. To Heather, I could write an entire library and it would not be enough to recognize everything you do for me. Love you always.

Finally, none of this would have been possible without the support provided to me by the DOD SMART scholarship program and by my home agency, SPAWAR Systems Center Atlantic in Charleston, SC. I am genuinely humbled by their belief in me and willingness to help support me in this endeavor.

This work was supported in part by SPAWAR System Center Atlantic TIKIBAR project number 2013-TIKI-030 and NSF contract number CNS-1213155. Portions of this dissertation also appear in [1, 2].

CHAPTER 1:
Introduction

Why is it that when one man builds a wall, the next man immediately needs to know what's on the other side?

George R.R. Martin, *A Game of Thrones*

Packet switching is one of the fundamental ideas behind the Internet. Any type of message that is to be sent from one application to another is broken down (i.e., packetized) into smaller chunks known as packets. Each packet that transits the Internet contains both data and one or more layers of control information known as headers. In the most basic sense, a packet's headers provide routing and control information that tell intermediate network devices where the data needs to go.

In addition to routing information, higher level transport headers carry a great deal of additional control information for connection-oriented protocols like the Transmission Control Protocol (TCP). In the case of TCP, the header provides for a number of extra features, such as connection negotiation, reliable stream reassembly, and flow control [3]. Each of these features was originally designed to operate on an end-to-end basis without assistance from the network. Traditionally, intermediate transit nodes were expected to ignore this information and simply focus on the transiting and routing portions of the headers [4].

In the modern Internet, packet transit is complicated [5] by a diverse abundance of network devices that violate this end-to-end principle—that intermediate network devices should not attempt to implement any functions that can be implemented "completely and correctly" by the endpoints [6]. We describe such devices in greater detail in Section 1.1, but for now note that they are able to introduce modifications to more than just the routing-related portions of a packet's headers. These modifications are possible because, aside from a few exceptions that will be discussed later, packet header information is rarely encrypted or authenticated [7] and thus transits the network unprotected.

Figure 1.1 illustrates a scenario where Alice wishes to send a packet to Bob through the

Internet. In the figure, Alice and Bob are endpoints. Alice is connected to the Internet through a series of devices on her network, connected to Bob's network through a path of routers in the Internet (designated by a cloud), and finally to Bob by the infrastructure in Bob's network. Any device along the path taken by her packets can view or alter her packets' headers as it deems fit.

As we will show, packet headers may experience a variety of changes to their fields while in transit, both intentional and unintentional. Intentional changes, such as network address translation desired by a network administrator, are an acknowledged fact of life on the Internet. Unintentional packet modifications, however, can be the result of misconfigurations or legacy devices and are an often under-appreciated issue that can have a widespread impact on the Internet and the evolutionary trajectories of the protocols that define it.

Through numerous examples documented by related measurement studies, we review in Chapter 2 exactly how such inadvertent packet header modifications have led to cases of performance issues, unintended protocol interactions, and blackholes where misunderstood packets simply get discarded by a network device. Unintended interactions can occur when a device misinterprets the meaning of a field, incorrectly alters it, and interferes with the correct operation of that protocol.

The primary motivation behind this dissertation is to develop the methodology and tools needed to have the most automated and generally usable platform as possible to further expose such problematic changes to critical packet header fields. The development of such a platform would impact a wide cross-section of Internet stakeholders, as discussed in Section 1.4.

Prior to this work, such a platform did not exist. The current state-of-the-art (described in Chapter 3) is an array of underused methods—all with some combination of deployment, incentive, or consistency issues that keep it from being used in the general case and preclude integration into TCP. Since TCP does not know anything about the networks it is going to traverse, it must make inferences about the end-to-end network state. Methods exist for inferring information such as congestion [8] and transmission errors [9], but none have yet been applied to detect when connection control information gets modified. If a platform could be developed that did not suffer from the same types of previously mentioned issues

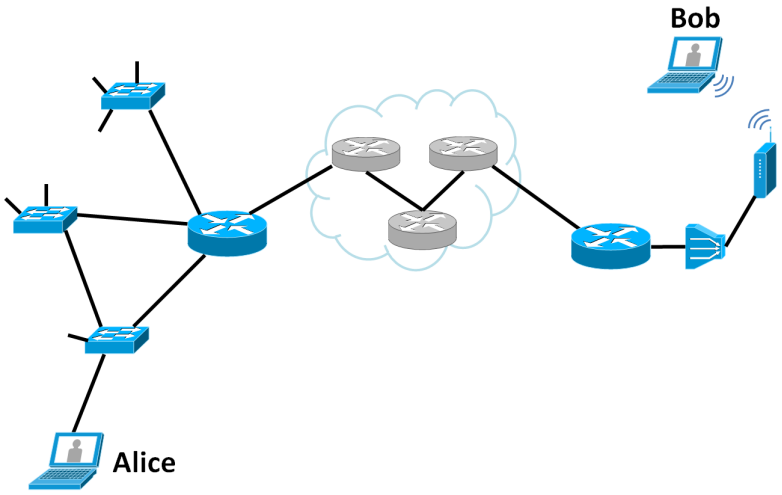

Figure 1.1: Simplified scenario of two hosts communicating over the Internet

(such as the solution advocated by this dissertation), it could fill that void and become another network state inference tool used by TCP.

1.1 Middleboxes

Finding the cause of inadvertent packet header modifications is difficult, since many different types of devices, in many different administrative domains, interact with a packet as it traverses the Internet. Traditionally, the primary objective of these devices was only packet forwarding (i.e., determining the next best hop to send the packet to based on information in a forwarding table). It is increasingly common, however, for packets to encounter devices whose primary task is something other than just the forwarding or routing of packets. request for comments (RFC) 3234 defines a term for these types of devices, *middleboxes*, and gives a taxonomy of the various types that existed at the time of publishing in 2002 [10]. The prevalence and diversity of middleboxes have only continued to grow since.

1.1.1 Types of Middleboxes

Some examples of well-known and commonly deployed middleboxes include: firewalls, network address translation (NAT) devices, performance-enhancing proxies, and transcoders that modify image files to reduce their size. Systems such as these are prevalent on the network, with each one having the ability to modify packets for its own purposes. A recent

study in 2012 showed that in networks of all sizes, the number of middleboxes is on par with the number of routers [11].

NAT devices, in particular, are extremely prevalent. Recent statistics from the network diagnostic tool Netalyzr show that about 90 percent of its sessions came from behind a NAT device [12]. Although Netalyzr's results likely suffer from a population bias, NAT has become nearly ubiquitous on home and corporate networks in large part due to the decreasing availability of globally-routable IPv4 addresses.

Also in use on the Internet are a myriad of other more specialized "security-enhancing" devices like sequence number randomizers [13], fingerprint scrubbers [14], active wardens [15, 16], and traffic normalizers [17]. Because these devices act as a man-in-the-middle (MITM), they can alter any bits within the entire packet header. NAT devices, for example, are expected to alter certain fields such as Internet Protocol (IP) addresses and TCP or User Datagram Protocol (UDP) port numbers, but nothing stops the same device from changing more header fields than that, such as sequence numbers and IP or TCP options. Any device acting as a MITM could also change payload data, but changes to payload data are already protected by other well-established solutions (e.g., Transport Layer Security (TLS)).

1.2 Problem Description

Middleboxes are difficult to manage and maintain. Networks of all sizes employ a diverse set of middleboxes that serve a variety of purposes. Even within the same network, the middleboxes are often from various vendors, usually run on separate physical hardware, and require configuration by a well-trained administrator. As a result, they can require a large support staff, which greatly adds to the already expensive cost of purchasing, licensing, and upgrading middlebox deployments [11]. In reality, the overly simplified communication scenario between Alice and Bob from Figure 1.1 looks more like the situation in Figure 1.2, where a variety of middleboxes permeate each network. And, on any given end-to-end path, problematic middleboxes may reside in an external administrative domain, complicating both efforts to debug and fix connection issues.

All of the previously mentioned issues with maintaining middlebox deployments contribute to the introduction of misconfigured, nonstandard, or out-of-date legacy behaviors in mid-

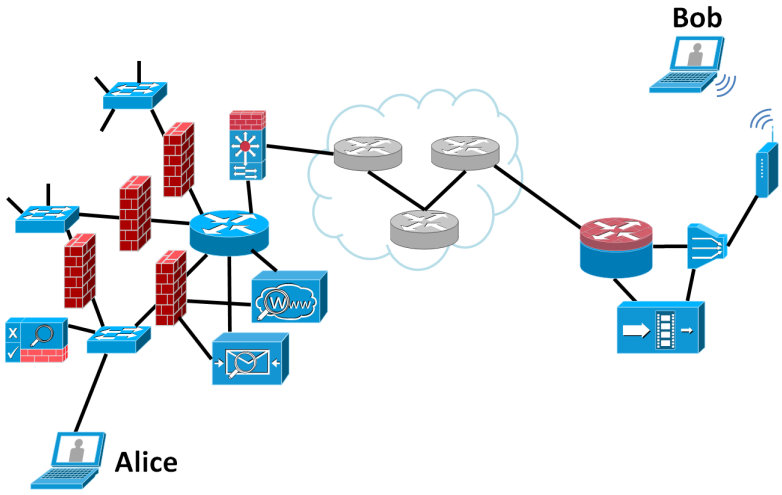

Figure 1.2: More realistic scenario of two hosts communicating over the Internet

dleboxes. In a survey of 57 network administrators, the majority overwhelmingly cited misconfiguration as the most common cause of middlebox failure, most likely due to the management and upgrade complexity involved [11]. As a result, the situation illustrated in Figure 1.2 appears much more daunting as the potential that Alice and Bob experience any performance or connectivity issues increases.

With their prevalence and how much power they have to alter packets and their semantics, it is a problem when middleboxes operate incorrectly or make unintended changes. As we will show in Chapter 2, such changes can result in unexpected protocol interactions and end-to-end performance issues. Furthermore, even just the threat of encountering such issues can negatively influence protocol innovation, forcing designers to scale back improvements and take overly conservative deployment strategies [18–22]. Examples can be seen in the designs of modern up-and-coming protocol extensions such as TCP Fast Open [23], Tcpcrypt [24], SPDY [25], and Multipath TCP [21] as their documentation discusses negative interactions with middleboxes. As we show in Chapter 2 and confirm in Chapter 8, such end-to-end traversal issues still occur and are a real problem on the Internet.

The existing solution space for this problem can be broken down into three primary categories of approaches: prevention, avoidance, or detection. Prevention involves using strong cryptography to stop middleboxes from tampering with packet headers. Avoidance tries to

fix the middleboxes themselves before issues arise. The final category, detection, includes solutions that check to see whether any modifications were made to the packet headers in-flight. Chapter 3 describes examples of related work that fall under each category and the limitations they have.

In this dissertation, we advocate a strategy of detection. We feel that this strategy is the most flexible, most cooperative with current middleboxes, and will be the most likely type of approach to achieve widespread adoption. Achieving widespread adoption is critical to any type of new protocol enhancement so that it is usable between a larger number of endpoint pairs. Since our design ultimately hinges upon adoption, we take steps throughout to make adoption-friendly design choices.

1.3 Summary of Contributions

The following list summarizes the primary contributions in this thesis:

1. Design of a novel methodology for the end-to-end detection of TCP/IP packet header manipulation: TCP-HICCUPS. The Handshake-based Integrity Check of Critical Underlying Protocol Semantics (HICCUPS) is an incrementally deployable extension to TCP that seeks to automate the question, *"Did my packets arrive at their destination with the same headers as they were sent with?"* As we will show throughout this dissertation, HICCUPS encompasses a unique set of design features, due to its specially tailored security model, that allow it to outperform the current state-of-the-art in areas such as general usability, efficiency, and in its ability to improve TCP:
    - HICCUPS is cooperative with currently deployed middleboxes, applying a tamper-evident seal to packet headers that allows devices to continue modifying packet headers as desired, but revealing that modification to a connection's endpoints.
    - HICCUPS is lightweight, requiring only a very small amount of processing overhead in our unoptimized kernel implementation that is on par with the overhead used for other packet processing functions in the Linux kernel.
    - HICCUPS operates in-band within TCP, requiring no out-of-band protocols such as Internet Control Message Protocol (ICMP). Not using any additional protocols helps improve the consistency of HICCUPS over current state-of-the-art approaches. As such, the inferences gathered by HICCUPS can be more

effectively used by TCP to maximize its performance in the presence of misconfigured or non-standard middleboxes.

2. Real-world implementation and testing of HICCUPS in a modern version of the Linux TCP networking stack. We have also provided a user space tool set that mimics the kernel behavior to allow for widespread and immediate testing.
3. Design and implementation of AppSalt HICCUPS, an optional extension that utilizes a novel cross-layer integrity protection scheme to discourage HICCUPS's integrity values from being modified in transit by certain middleboxes, even after the HICCUPS integrity algorithm becomes public knowledge.
4. Deployment of, and measurements from, HICCUPS across 26,304 diverse directed port/path pairs on the live Internet.
5. Documented instances of degenerate middlebox behavior and the ways in which HICCUPS cooperation could improve transfer performance.

1.4 Impact on Relevant Stakeholders

The significance of this work is the impact it could have on our understanding of the global Internet architecture. Successful adoption of this technology would not only make debugging and troubleshooting easier, but yield new insights about the integrity of packet transit across a large portion of the network. It would also enable interesting future measurement studies that could clarify the impact of middleboxes and allow network administrators to implement new protocols more safely, without alienating a small set of users behind a faulty middlebox. This could accelerate the deployment of new protocols that enhance robustness, reliability, or offer new functionality.

Several incentives exist to promote adoption of our proposed solution. We envision HICCUPS benefiting all groups of key Internet stakeholders:

- End users and content providers want to know that their traffic is treated fairly by their transit providers (e.g., in the context of network neutrality [26]).
- Protocol designers want to know how their new designs will be affected by network middleboxes.
- Large Internet companies want to take advantage of new protocol extensions designed to increase performance, but do not have a means to tell if the extension

would be safe to use.

- System and network administrators would have access to a new diagnostic tool to troubleshoot more complex connectivity issues.
- The networking community would be able to draw upon a wealth of new information for measurement studies to better understand the global Internet.
- Any end host enabled with our detection code could instantly become a cooperating end point in a path integrity test. This would be similar to how `ping` and `traceroute` are used today to enable network tests with a large number of hosts without requiring prior coordination with the system administrators.

1.5 Document Structure

The remainder of this dissertation is organized as follows:

Chapter 2 provides background information to fully explain the various network mechanisms discussed in this work, as well as related research on middleboxes and the issues they can cause.

Chapter 3 begins with a survey of the various existing solutions introduced in Section 1.2 and then synthesizes their shortcomings into a set of well-defined architectural design principles. These goals are then used to evaluate our methodology in the context of various relevant works.

Chapter 4 discusses methods for transmitting integrity information and the implications of various design considerations.

Chapter 5 presents our design for HICCUPS and describes its methodology in detail.

Chapter 6 considers what can happen when a middlebox attempts to fake the integrity information and details AppSalt, an optional extension that adds protection to the integrity values transmitted by HICCUPS.

Chapter 7 details our implementation in the Linux kernel and related design choices.

Chapter 8 presents the results from experiments using our protocol on the Internet.

Chapter 9 concludes our work by summarizing key points and discussing opportunities for future work.

CHAPTER 2:
Background and Impact of Middleboxes

Intermediaries make the web a hostile place for protocol changes.

Mike Belshe, designer of the SPDY protocol

In this chapter, we analyze the impact of misconfigurations and legacy behavior in middleboxes on both the performance and evolutionary trajectory of the Internet. Prior to discussing such issues with middleboxes, Section 2.1 covers relevant background information to help setup and more thoroughly understand how the affected protocols operate. Then, building upon the issues introduced in Section 1.2, we more closely examine the breadth of disruptions that can arise from misconfigured middleboxes (Section 2.2), going into depth on several selected issues (Sections 2.3–2.4). Finally, we conclude with a discussion of the impact that these disruptions have on overall network security (Section 2.5) and on protocol innovation going forward (Section 2.6).

2.1 TCP/IP Background

TCP/IP is the primary suite of protocols on which the Internet operates. The various elements of communication on the network are broken out into four layers which make up the full protocol suite. In order from highest to lowest, the layers are: application, transport, network, and data link [27]. Each layer is responsible for a different aspect of communication and has its own set of associated protocols. Figure 2.1 graphically represents the ordering of the layers for the TCP/IP model along with some exemplary protocols.

The application layer is the most flexible with respect to what bit structure the network will tolerate and transmit. The TCP/IP model treats any contents in the application layer portion of a packet as application data. Other popular network models such as the Open Systems Interconnection (OSI) model [28] further subdivide this layer, but we do not describe them here since the additional elements are out of scope; the work covered in this dissertation does not deal with any application layer elements. Traditional end-to-end philosophy holds

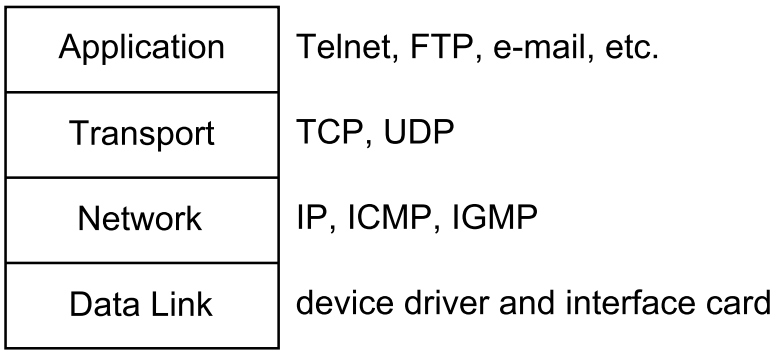

Figure 2.1: The four layers of the TCP/IP protocol suite (from [27])

that the network should completely ignore this layer and let end hosts handle any processing of the application layer [6].

The transport layer provides port-based multiplexing services, as well as an optional set of end-to-end services that can help improve an application's performance (e.g., flow control and reliable stream reassembly—ordering and loss correction). Both UDP [29] and TCP [3] are provided by the TCP/IP protocol suite, with TCP being the one that provides the optional services just mentioned. We describe TCP in more detail in Section 2.1.3.

The network and data link layers help network devices get a packet from one end host to another. These lower layers have considerably less context that is maintained end-to-end. Figure 2.2 shows an example of how a packet may look as it crosses the wire. This particular packet is a Hypertext Transfer Protocol (HTTP) `GET` request. Web browsers send this type of request in order to fetch a webpage from a remote webserver [30]. In the figure, each byte of the packet is shown as a pair of hexadecimal digits. Note the construction how each header is layered over the original application data (the `GET` request).

In the remainder of this section, we describe selected protocols and features that are relevant to the background middlebox work presented in the remaining sections of this chapter.

2.1.1 IP

The structure of an IPv4 header is shown in Figure 2.3. Working from the top line down, IP defines some control information about each packet, a pair of 32-bit addresses that represent an interface on a particular end host, and an options block. The control information

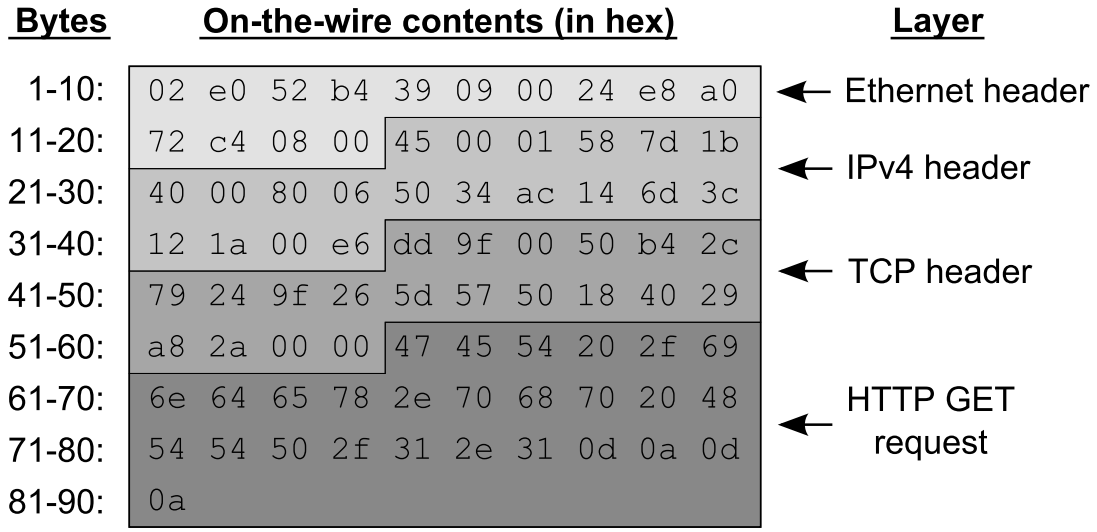

Figure 2.2: Example of what a packet looks like on the wire (this specific packet is an HTTP request to get a webpage from a webserver)

held within the first 12 bytes include a version number, the length of the IP header, the differentiated services code point (DSCP), Explicit Congestion Notification (ECN), length of the total packet, fragmentation-related information, a hop expiration counter, the type of transport protocol being carried, and a 16-bit checksum of the header.

It is important for later to note the history with the DSCP and ECN fields. Before DSCP and ECN were standardized in 1998 and 2001, respectively, the entire second byte of the IP header (shown in the top row of Figure 2.3) was defined as the type of service (ToS) field. RFC 791, the specification for IP since 1981, shows the ToS field being used by gateways to hold indicators marking service and priority requirements for packets. When ECN was created, the semantics of the highest two bits in that second byte were redefined. No longer were they "reserved for future use" and to always be set to zero, but now possess meaning. ECN is described further in Section 2.1.3.

The field redefinition issues experienced with ECN were not a singular event. Other proposals are currently under consideration by the Internet Engineering Task Force (IETF) to reclaim other bits of the headers. Some features that were useful in the early years of TCP/IP design are rarely used in the modern Internet and may be subject to reuse. Some examples are fields related to fragmentation [34] or the TCP urgent pointer [35].

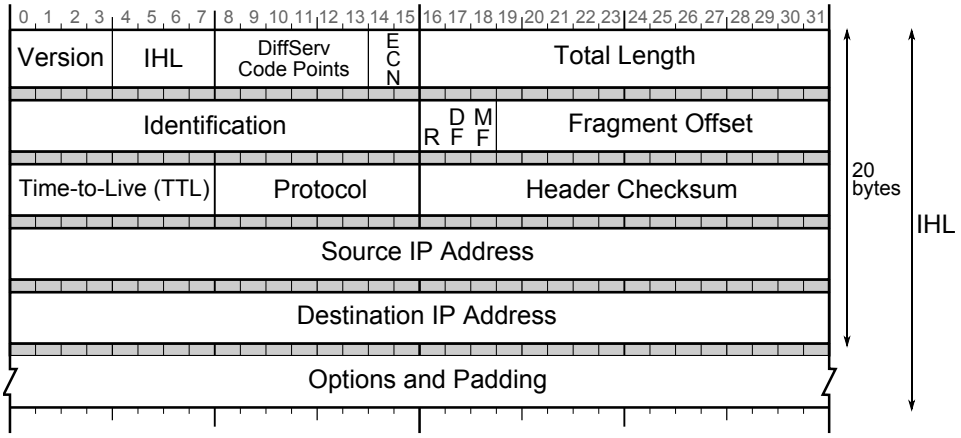

Figure 2.3: Structure of an IPv4 header (after [31–33])

2.1.2 ICMP

ICMP [36] is primarily used to transmit status and error messages between systems on the Internet. For example, when a UDP datagram is sent to a port that is closed, the receiving host will respond with an ICMP *destination port unreachable* message to let the sender know that the port was closed. There are many different types of ICMP messages, which all use the base format shown in Figure 2.4. The format of the latter portion of the ICMP header depends on which type of message it is. Each message is defined by a type and a subtype, also called a code. For example, the destination port unreachable message is type three, code three. We next discuss two specific messages that will be referenced later in this document.

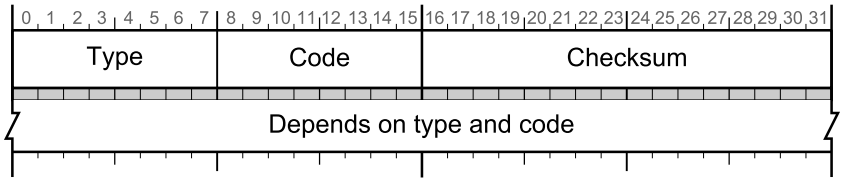

Figure 2.4: Base structure of an ICMP header (after [36])

TTL exceeded in transit

The *time-to-live (TTL) exceeded in transit* message is a subtype of the time exceeded message (type 11, code zero). This message is an error code sent by a router when it receives

a packet with a TTL value of one. After the router performs its decrement, the TTL is zero and the packet is considered to be expired. This particular message provides the basis for a number of path diagnostic tools (e.g., `traceroute` [37], Paris Traceroute [38], Tracebox [39], etc.).

`traceroute` was originally developed by Van Jacobson in 1987. Its intuition is simple: starting with a TTL value of one, send a succession of packets with singly increasing TTL values. The packet will expire at each successive hop along a path and the sender will be able to enumerate the routers along the path by the ICMP TTL exceeded messages that come back in response.

Fragmentation needed and DF set

The *fragmentation needed and don't fragment (DF) set* message is a special subtype of the destination unreachable type (type three, code four). This message is an error response to a packet that is too large to traverse a link and cannot be fragmented because it has the DF flag set in its IP header, indicating that it is "not to be internet fragmented under any circumstances" [31]. Since the packet cannot be fragmented by the router, the router drops the packet and sends the resulting ICMP error message back to the packet's sender.

This message is used by a technique to automatically discover the largest acceptable packet size that a path will accept: path maximum transmission unit discovery (PMTUD). PMTUD was developed as a means of being able to eliminate the act of transparent in-flight fragmentation, which by 1987 was widely considered to be too inefficient, too complex, and too much of a performance hit to be worth supporting [40]. By the mid 2000s, transmission rates had largely exceeded the capacity allowed by the 16-bit IPID field to differentiate between fragments, further condemning the practice of transparent in-flight fragmentation [41].

PMTUD works best when routers implement an accompanying change to their type three, code four messages introduced by RFC 1191: the definition and population of the Next-Hop MTU field. When it is the case that the MTU-restricting routers along a path implement this improvement, the process of PMTUD is as shown in Figure 2.5. If a router does not provide such feedback, it is left up to the host to "search" the MTU space.

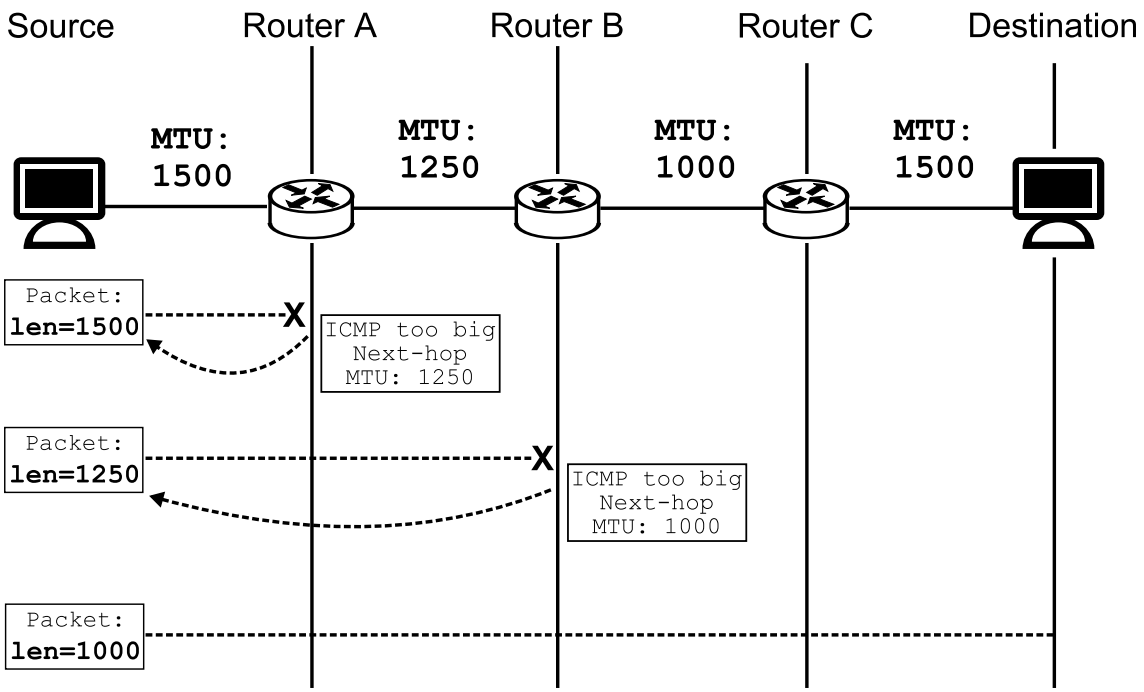

Figure 2.5: Example Path MTU Discovery scenario

In practice, a sizable number of occurrences of firewalls blocking the return of the ICMP feedback messages prevents PMTUD from operating properly. In a 2004 measurement study, Medina *et al.* found a mere 41 percent success rate of PMTUD where blocked ICMP messages were pinpointed as the presumable cause of failure for 18 percent of the servers tested [42, 43]. Luckie and Stasiewicz revisited the issue in 2010 to find a higher PMTUD success rate of 78–80 percent, nearly double that of the the 2004 result, but found other causes of PMTUD failure, such as a software bug where the DF flag was still being set for very small MTU values [44].

Ultimately, however, ICMP blocking and issues integrating with TCP [45] were so problematic that PMTUD was rewritten to use TCP instead of ICMP [46]. The issues that PMTUD experienced with ICMP blocking are a notable example of issues with out-of-band feedback mechanisms, and as such, have highly influenced the design of our methodology. We elaborate more on this in Section 4.4.

2.1.3 TCP

The structure of a TCP header is shown in Figure 2.6. Working from the top line down, TCP defines: port numbers used for connection multiplexing, sequence and acknowledgment numbers used for stream ordering and reassembly, flags used for connection negotiation and teardown, a window size used for flow control, a 16-bit checksum and urgent pointer, and an options block that designers included so that the protocol could be expanded later. We explain each of these features in more detail in the following subsections.

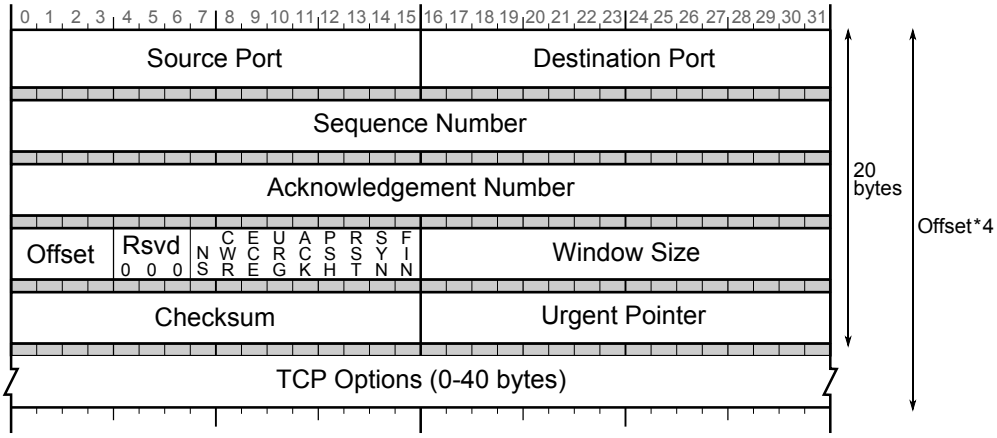

Figure 2.6: Structure of a TCP header (after [3])

Connection negotiation

TCP is a connection-oriented protocol: all data is transmitted within the context of a connection between a pair of endpoints. New TCP connections are created by a process known as the three-way handshake (3WHS) which is illustrated in Figure 2.7. In the figure, Alice is the endpoint that wishes to initiate the connection. To do so, she sends Bob an empty TCP segment with the synchronize (SYN) flag set. This action triggers the start of the 3WHS. If Bob has an application socket listening on the destination port, he responds with a SYN of his own with the acknowledgment (ACK) flag also set. To complete the handshake, Alice ACKs Bob's SYN/ACK. Connections can be closed gracefully via the finish (FIN) flag, or abruptly via the reset (RST) flag.

Sequence numbers

In TCP, all application data is sequenced by byte so that it can be reliably transferred from one end host to another. TCP ensures that all data is both complete and in the proper order

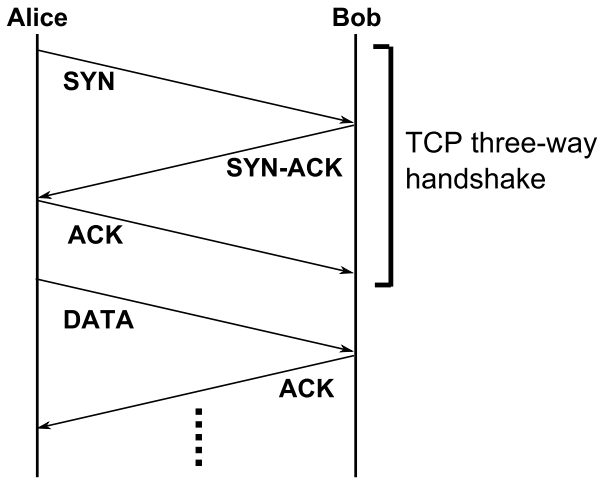

Figure 2.7: Example of connection negotiation in TCP using a three-way handshake between two endpoints, Alice and Bob)

before handing it up to the application layer. One TCP informs another about the portion of the data stream that it has cumulatively intact by acknowledging the next byte that it needs in the acknowledgment number field.

Figure 2.8 illustrated an example TCP session with simplified sequence and acknowledgment numbers. When Alice initiates the connection, she chooses an initial sequence number (ISN) for her SYN. Bob acknowledges her ISN by setting his acknowledgment number to the number of the next byte he expects to receive from Alice. Since TCP is bidirectional, Bob also chooses an ISN to represent the flow of bytes from him to Alice.

In the example in the figure, one of Alice's data segments gets lost on its way to Bob. TCP's usage of sequence numbers informs Bob of the lost data when the next packet he gets from Alice contains data beyond what he was expecting to receive next. Since Bob is missing a portion of the data stream, he must send a duplicate acknowledgment to Alice. One can already see how this can be inefficient if only one packet is lost and none of the successively received bytes can be acknowledged. This problem is solved by selective acknowledgment (SACK) which is discussed later with the TCP options.

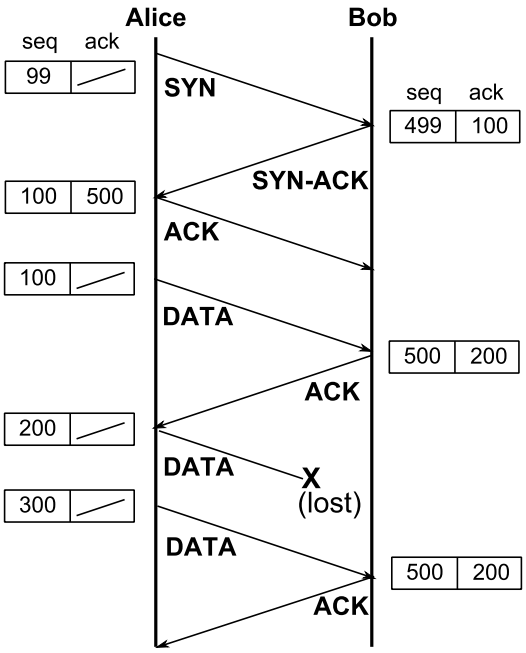

Figure 2.8: Example of sequence number usage in a TCP connection between two endpoints, Alice and Bob)

Flow Control

Each TCP implementation maintains a receiver buffer that acts as a temporary hand-off area between TCP and an application. As new data arrives, it is queued in the buffer until the application makes a system call to retrieve it. TCP will only allow completely reassembled stream content to be read by an application, holding in the receive buffer any content that is not in the correct order or is missing a piece of data [3, 47].

The flow control window in TCP is used to keep a sending TCP from overwhelming a receiver's buffer capacity. The flow control window, also called a receive window, represents the maximum number of bytes that a TCP is willing to accept at a given time. The field is 16 bits in length, but can support larger window values through the scaling option discussed later in this section.

Consumption of the data in the receive buffer can be limited by an application, the system's CPU usage, or by TCP's stream reassembly behavior when data loss is experienced [47]. For the last cause, the worst case scenario is that a full window of data is sent from one

TCP to another, but the first packet of the window is dropped. Since TCP must reliably reassemble the data stream, the buffer on the remote TCP is left mostly full and there may not be enough buffer capacity to accept new packets, resulting in wasted bandwidth and retransmissions. If a TCP flow is throttled back due to the flow control window, it is considered to be *receive window-limited*.

ECN

Traditionally, a TCP sender must rely solely on timeouts or lost segments in order to infer when a path is congested. ECN [33] is an improvement that allows routers to more actively assist in congestion control by marking packets as local buffer pressure increases. Indications from routers signal the receiver to alert the sender to reduce his rate before the router will be forced to drop packets, which is more inefficient than having the sender preemptively slow down.

At the bit level, the ECN semantics are transmitted via several fields within the IP and TCP headers. The congestion mark is made by setting a pair of bits in the IP header, shown in the middle of the top row of Figure 2.3. The two-bit field can take on three different meanings as shown in Table 2.1. When a router wants to mark congestion, and the packet is ECN-capable, the router changes the ECN-capable transport (ECT) code point in the IP header to the congestion encountered (CE) code point. A receiver that gets a packet with the CE code point set knows that congestion occurred and should tell the sender to reduce its rate. Feedback notification from the receiver to the sender is done by setting the ECN echo (ECE) flag in the TCP header of returning acknowledgment packets until the sender gets the message and reduces its congestion window. The sender can then inform the receiver that it has indeed reduced its window by setting the congestion window reduced (CWR) flag of the TCP header. Both flags are shown on the left side of the fourth row in Figure 2.6.

It is important to note that all of the bits occupied by these ECN flags had other meanings before ECN was standardized in 2001. The DiffServ and ECN code points fields shown in the top row of Figure 2.3 were originally a single byte known as the ToS field. The TCP flag bits shown on the left side of the fourth row of Figure 2.6 were listed as reserved and expected to always be zero. An experimental enhancement to ECN adds semantics to another bit in the TCP header, the nonce sum (NS) bit [48].

Bits	Code Point	Meaning
00	Non-ECT	Not ECN-capable
10 01	ECT	ECN-capable, no congestion
11	CE	Congestion encountered

Table 2.1: ECN code points in the IP header

Options

The original designers of IP and TCP wanted to allocate space within the headers to allow for future expansion and features. As a result, both IP and TCP allow their headers to be extended up to 40 additional bytes in length for placement of consistently formatted options. Since then, many enhancements that use this space have been proposed and several have become standards. Some of the more well-known standardized TCP options include:

- **Maximum Segment Size**: sent in the options block of the SYN; defines the largest sized segment that should be sent as part of that unidirectional flow [49, 50].
- **Selective Acknowledgment**: if its use is negotiated in the 3WHS, receivers can additionally acknowledge blocks of data received beyond the cumulative stream boundary [51]. An example of SACK operation is shown in Figure 2.9.
- **Window Scaling**: sent in the options block of the SYN; defines a power of two scaling factor to be applied to that unidirectional flow's receive window [52].
- **Timestamps**: contain two four-byte long timestamp values and can be attached to any packet of a TCP connection to help stacks measure round trip times (RTTs). One of the timestamps is simply an echo of what was received from a remote TCP in an ACK [52].
- **Multipath TCP**: allows TCP to utilize multiple paths as part of a single connection in order to maximize efficiency of resource usage or swap between different types of networks such as Wi-Fi and cellular data networks [21].

2.2 General Measurement Studies

Architectural issues with middleboxes and their unintended consequences have long been documented [18, 19, 42, 43]. In 2004, Medina *et al.* detailed several issues caused by unexpected interactions on the part of a middlebox [42, 43]. In addition the issues with PMTUD previously described in Section 2.1.2, middleboxes were found to disrupt a number of other

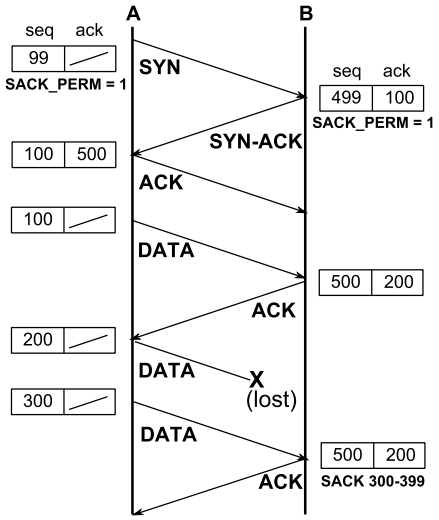

Figure 2.9: Scenario showing how SACK helps improve performance

protocol features. For example, the usage of a non-standard IP option resulted in failure of over 70 percent of connections, with mixed results even for common options. The concern of such behavior is not just that the options were not propagated, but even worse that the entire packet was dropped at some point along the path (i.e., the "blackhole" problem). The authors also found that ECN was a problem area as well, where middleboxes overwrite flags, causing failures with negotiation and congestion notification. We look at ECN issues more closely in Section 2.3.

Honda *et al.* used measurements taken by their tool, TCPExposure, to examine how TCP options are treated by middleboxes [19]. They found instances of TCP options, both known and unknown, being stripped from packets, sequence numbers being translated, and even some port-specific behaviors where options were stripped on a random high port, but not on port 80. Middleboxes along some paths were also found to be very fragile dealing with out-of-order data. Ultimately, they found that at least 25 percent of the paths seen in the study had a middlebox whose behavior depended on the transport-layer (e.g., TCP) of packets that passed through the middlebox. Not only is this interference detrimental to the validity of the protocol interactions, but it is also difficult to diagnose and makes troubleshooting a complex endeavor.

2.3 Explicit Congestion Notification

ECN [33] is an interesting TCP/IP enhancement worth closer examination as experiences with ECN distill the essence of the middlebox problem. Recall from Section 2.1.3 that ECN relies on a delicate series of cross-layer interactions to function properly:

1. Routers must be able to mark packets (at the IP layer)
2. The receiver must echo the congestion mark back to the sender (at the TCP layer)
3. The sender must properly acknowledge and slow down

There are many opportunities for the above series of interactions to be disrupted. Not only must the three parties involved (sender, receiver, and routers) properly follow the protocol, but any middleboxes along the path must retain all of the ECN semantics. For example, suppose a middlebox inadvertently clears the CWR flag, keeping the sender from acknowledging the receiver's ECE and letting it know that the sender has reduced its congestion window. The sender would continue to see the ECE flag set on acknowledgments returning from the receiver and not realize that the receiver never saw the sender's CWR acknowledgment. As such, the sender would infer that congestion was still occurring on the path and continue to reduce the size of its congestion window.

2.3.1 Issues with legacy devices and ECN

In addition to the complex series of interactions, issues with ECN are further exacerbated by the fact that every one of the header fields used by ECN has held other meaning. At least over 30 years went by between the first documentation of TCP and IP before ECN was devised and standardized in 2001. Each of the fields redefined by ECN is an opportunity for a legacy middlebox to misinterpret packet header bits and potentially disrupt the ECN interactions.

While the study by Medina *et al.* found occurrences of ECN-blocking middleboxes, it was performed in 2004 when ECN was fairly new and had not yet achieved widespread implementation. For example, 93 percent of the servers they tested did not even support ECN. Seven years later, in 2011, Bauer *et al.* revisited ECN readiness in Internet hosts and found that even though many servers were capable of using ECN, a non-trivial number of problems with middleboxes disrupting the ECN fields still existed [20]. The most common

problem involved treating the 6-bit DSCP field and the 2-bit ECN field as the old 8-bit ToS field. Attempts to overwrite or clear what the network device thought was the ToS field resulted in accidentally overwriting the ECN information.

Such overwriting can also impact connection performance. For instance, if any of the congestion signaling bits are inadvertently set when there was not any congestion to begin with, performance will suffer. As stated before in Section 2.3, ECN relies on a delicate series of cross-layer interactions to properly communicate when congestion was experienced and when that communication was acted upon. Any spurious overwriting of its fields can corrupt the state between the two endpoints, confusing ECN and impacting performance.

It is important to note that all of these issues are unintentional. None of the disruptive issues noted above and in the studies by Medina *et al.* and Bauer *et al.* were due to an active adversary attempting to harm the performance of connections. They were due to poorly configured switches or routers that have not been updated in years. In the case of Bauer *et al.*, when the authors notified the network operator for one of the issues they detected, the operator was unaware of the issue [20]. Examples such as with ECN serve to strengthen the argument that the model of the inadvertent adversary, even though under-appreciated when compared with an actively malicious adversary, can still be a greatly disruptive force on the Internet and a threat to overall network stability.

2.4 ISN translation and SACK

Another example of an Internet protocol being manipulated by a middlebox is found in the manner in which some firewalls implement TCP ISN randomization. ISN randomization is done by some firewalls to protect hosts behind them that insufficiently randomize their ISNs, leaving the values predictable and making the connection vulnerable to spoofing attacks and off-path resets (see Section 4.7.6 for more). Once an initial sequence number is changed, all sequence numbers must continue to be translated throughout the life of the TCP connection. However, in many cases this feature does not properly translate SACK values and passes the untranslated SACK blocks with translated ISNs (see Figure 2.10). Enabling this feature harms both performance and overall throughput since the out-of-window SACK blocks only confuse TCP and do not reduce any retransmissions. Documentation from one leading vendor now recommends disabling the module in their firewall [13].

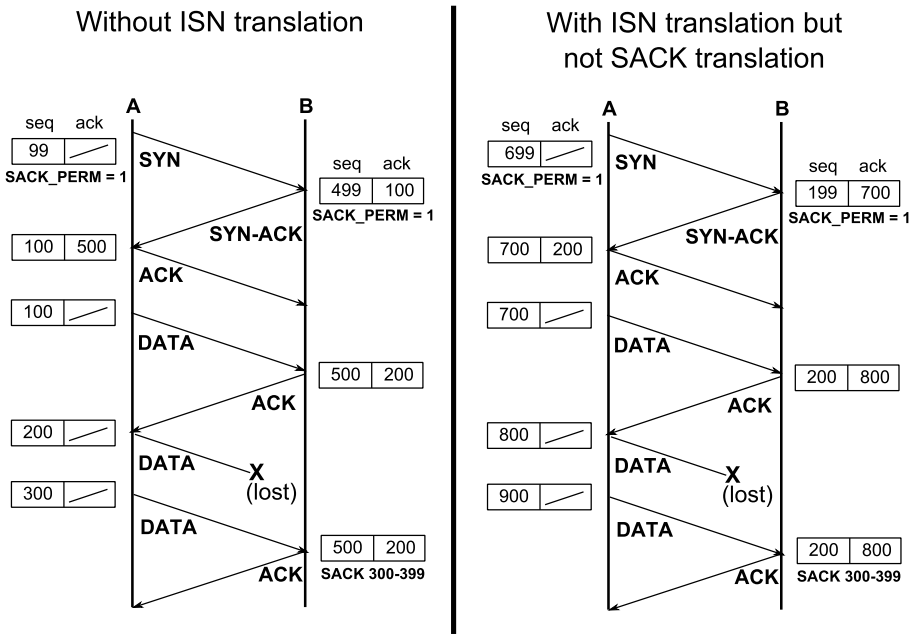

Figure 2.10: Scenario showing how SACK can be disrupted by poor implementation of ISN translation

Problems such as a mismatch between SACK blocks and overwritten sequence numbers can be very difficult and time consuming to identify and fix. We encountered this very issue ourselves on our own organization's network. In order to diagnose it, we required a cooperating endpoint remotely located outside our network to perform low-level comparison between traffic sent and traffic received. In addition to requiring resources outside one's administrative control to troubleshoot a problem within their own network, this problem is very subtle and requires the keen eye of a trained administrator to recognize and understand the issue. With the methodology we present in this work, the detection logic we build into TCP would alert both the user and the network stack itself to such issues, eliminated the need for a skilled administrator to troubleshoot.

Instances of SACK mismatches due to middleboxes are found in the academic literature as well. Honda *et al.* gives a general warning about sequence numbers being included in various TCP options due to the fact that they are often inconsistently overwritten by middleboxes [19]. Honda also notes that this issue could even become worse if any of the various proposals to expand the TCP options space [35] are ever adopted. Extending the

options across multiple packets could make copies of the sequence numbers even harder for a middlebox to locate and translate. Hesmans *et al.* further discuss the issues with SACK and suggest a change to how out-of-window SACK blocks are interpreted [53].

2.5 Negative impact on overall network security

As seen with ECN and other extensions, misconfigurations and legacy behavior in middleboxes can stifle innovations that were designed to add features and make the network more robust. Table 2.2 summarizes some of these examples.

In the case of ECN, for example, it was designed to enhance congestion control in TCP/IP. By integrating state from routers, the network can provide added functionality such as early congestion detection to increase network efficiency and fairness. When middleboxes inadvertently disrupt the interactions of the ECN fields, those gains can be lost and the potential is there to completely break congestion control itself as shown by example in Section 2.3. In addition, on paths where congestion marks or echoes are always falsely set, performance will be severely degraded. Such scenarios can function almost like a denial of service (DoS) attack in the sense that an end host's performance can be severely degraded by the inadvertent modifications.

In general, the IETF and various network administrators, especially those with large user bases, are far more reluctant to enable these new extensions when unexpected protocol manipulations are taking place and causing connections to fail. With respect to ECN in particular, the feedback we received from one large content provider was that "we want to enable ECN, but do not because enabling ECN may adversely affect some of our users." [58]

This reluctance, induced by a small number of bad paths, can negatively impact the overall security and stability of the Internet as usage and needs evolve, bringing with them the need for new extensions to core network protocols. In this sense, middleboxes and other systems that cause these problems are *inadvertently adversarial*. This differs from the typical adversarial model in that while a system is not intentionally malicious, it can and does cause unforeseen problems through its modifications of packet fields.

Protocol Feature	Reason for Disruption	Issue	Impact	Source(s)
Path MTU Discovery	Legacy, Policy	ICMP blocking	Degraded performance	[43, 44]
IP Options	Legacy, Performance	Blocking, option stripping	Blackholes, poor extensibility	[18, 43]
TCP Options	Legacy, Policy	Option stripping	Poor extensibility	[19]
ECN	Legacy, Misconfiguration	Blackholes, mark concealment, improper congestion signals	Degraded performance, attack congestion control	[20]
SACK	Misconfiguration	Out-of-context SACK numbers	Degraded performance	[19, 53]
Receive Window	Policy	Artificially-limited window size	Net neutrality	[54–56]
Window Scaling	Misconfiguration	Option modification	Degraded performance	[25, 57]

Table 2.2: Examples of Middlebox Interference

2.6 Impact on Protocol Innovation

A more complex and broader impact of protocol interaction issues described in this chapter is their impact on the design of new protocol extensions. Fear of compatibility issues with middleboxes is a constant source of stress for protocol designers. Constantly in fear of their design working incorrectly for a small number of paths, designers are forced to specifically account for these issues in their designs.

Numerous examples of middlebox impact can be found in the literature of modern up-and-coming protocol extensions such as TCP Fast Open [23], Tcpcrypt [24], SPDY [25], Multipath TCP [21], and Gentle Aggression TCP [59]. In each work, the authors describe specific considerations taken in their designs to be compatible with both known and unforeseen issues with middleboxes. Often, a note about their efforts is even included in the abstracts of their works, and in the case of RFC 6824 for Multipath TCP, an entire section

covers “Interactions with Middleboxes [21].”

Protocol designers’ frustration with middlebox compatibility issues is growing and research towards understanding and/or solving these issues has been very active since 2010, when many of the newest generation of protocol innovations started appearing. The growing frustration is even apparent in the publication trends of many of the contributors to these new protocols, as they alternate between designing their protocol, and understanding and solving issues with failed middlebox interactions. Specifically of note, many of the designers involved with Multipath TCP (e.g., Raiciu, Handley, Bonaventure, Honda, Paasch, and Detal) have also become involved with various middlebox-related initiatives [19,39,53,60].

Perhaps the best summation of the current environment toward protocol innovation on the Internet is a note given in a presentation by Mike Belshe, designer of the SPDY protocol, in 2011: “intermediaries make the web a hostile place for protocol changes” [25]. The situation is not completely grim, however. In the following chapter, we will discuss various solutions that can be applied to help understand, mitigate, solve, or prevent broken middleboxes from inadvertently interfering with new protocol extensions, as well as introduce our novel solution to the field, TCP HICCUPS.

CHAPTER 3:
Solutions for Middlebox Issues

If I have seen further it is by standing on the shoulders of giants.

Sir Issac Newton

Surveying the possible solution space for the middlebox-related issues described in Chapter 2, we find that the current applicable state-of-the-art falls under one of the three categories of approaches mentioned in Section 1.2: *prevention*, *avoidance*, and *detection*. As we will show in this chapter, each currently available solution suffers from some key limitations that reduce its efficacy within the context of our problem domain.

Beginning in Section 3.1, we examine network traffic integrity mechanisms that can prevent successful packet header tampering. Such mechanisms treat middleboxes as simply another MITM adversary and make it so that if a middlebox does modify a packet it will not be accepted and processed by the remote endpoint. We also discuss the complications involved in applying the necessary cryptographically-strong integrity (i.e., using computationally secure encryption and hashing algorithms such as AES [61] and SHA [62], respectively) to packet headers, emphasizing that application of traditional integrity mechanisms to packet headers is not a problem with a simple, straightforward solution. Middlebox traversal and key distribution issues commonly plague such cryptographically-strong approaches.

Next, in Section 3.2 we discuss works that approach middlebox coordination from a unique and novel angle by attempting to overhaul the middlebox architecture using elements of software-defined networking (SDN). Concluding our coverage of the solution space, in Section 3.3 we take a closer look at various detection-based solutions. One such detection-based solution that is particularly applicable to our domain and merits closer inspection is an out-of-band diagnostic tool known as Tracebox [39]. Tracebox is designed to detect in-path middlebox modifications in a cooperative environment, but its lack of integration into TCP leads to several limitations that we further describe in Section 3.3.3. A summary of all three categories, along with pointers to their corresponding sections, is shown in Table 3.1.

Section	Paradigm	Description	Examples
3.1	Prevention	Stop middleboxes from successful tampering	IPsec, tcpcrypt
3.2	Avoidance	Overhaul and streamline middlebox architecture to eliminate issues	SIMPLE, APLOMB
3.3	Detection	Detect and, if possible, workaround middlebox tampering	checksums, Tracebox

Table 3.1: Categorized breakdown of the current solution space for inadvertently adversarial middlebox packet header tampering

After cataloging the space of applicable solutions, we construct a set of architectural design requirements in Section 3.4 based in part on the shortcomings and limitations of current solutions. We hypothesize that a solution adhering to these requirements could detect packet header modifications and provide a way for TCP to deal with issues caused by middleboxes. By addressing a different point in the design continuum, our solution can advance the state-of-the-art and actively help improve TCP. Finally, we provide a comparative overview of our solution's high-level properties as compared to the range of existing possible solutions.

3.1 Tamper Prevention

Tamper prevention schemes use cryptography to prevent a packet header from being modified and then accepted by an endpoint. If a host receives a packet that does not match the corresponding scheme's integrity checks, it discards the packet and forces TCP to retransmit. The security model used in the designs of these schemes assumes the following:

- An in-line device, M, that can see all traffic between two endpoints, A and B
- M can arbitrarily inject new packets and make them appear to originate from either A or B [63]
- M can modify packets or their headers and recompute any public-knowledge checksums
- M can reorder packets
- M can discard certain packets
- M can usually see and affect all of either A's or B's traffic

Due to their location within the network, middleboxes have all of the capabilities of M in the above security model. As a result, the tamper prevention schemes discussed in this section

will prevent a network device middlebox from successfully modifying packet headers, and could be applied to our problem of stopping inadvertent header modifications resulting from misconfiguration and legacy behaviors. However, as we will show at the conclusion of this section (in Section 3.1.3), many of these schemes are overpowered when it comes to addressing the issues discussed in Chapter 2 due to the strength of the assumptions in their security models. A more targeted approach would be more efficient and not as cumbersome to deploy and use.

3.1.1 Pre-shared keys and PKI

In addition to utilizing computationally secure encryption and hashing algorithms, the most secure of the tamper prevention schemes perform trusted key distribution for protected authentication. The threat in not using authentic keys (i.e., either cryptographically signed or distributed out-of-band) is that a particularly aggressive middlebox can perform a MITM attack on the conversation by inserting itself between the two parties' communications as keys are negotiated. The MITM could negotiate keys with each endpoint individually and act as a proxy between them, gaining full read and write access to the conversation. In this section, we discuss such solutions that provide the strongest levels of security: Internet Protocol Security (IPsec), TCP MD5, TCP Authentication Option (TCP-AO), and Secure Sockets Layer (SSL)/TLS.

IPSec

IPsec is a suite of security enhancement protocols that operate on top of the IP layer [64]. Two major components of IPsec are the Encapsulating Security Payload (ESP) [65] and Authentication Header (AH) [66], which can be applied alone or in combination. Both technologies guarantee connectionless integrity and data origin authentication of packets through the use of computationally secure cryptography and digitally signed keys. ESP additionally can provide payload confidentiality.

Each ESP and AH can operate in one of two possible modes: *transport* or *tunneled*. In transport mode, the corresponding IPsec headers are simply layered in between IP and the header for whatever transport layer protocol is in use. In tunneled mode, the packet to protect is encapsulated within a new IP and IPsec packet. ESP only protects itself and the IP payload, so it can only protect IP header fields when operating in tunneled mode—the use

of AH would be required to protect the outermost layer of IP header fields. Figure 3.1 illustrates what a packet may look like once it has been applied with the AH in both transport and tunneled modes.

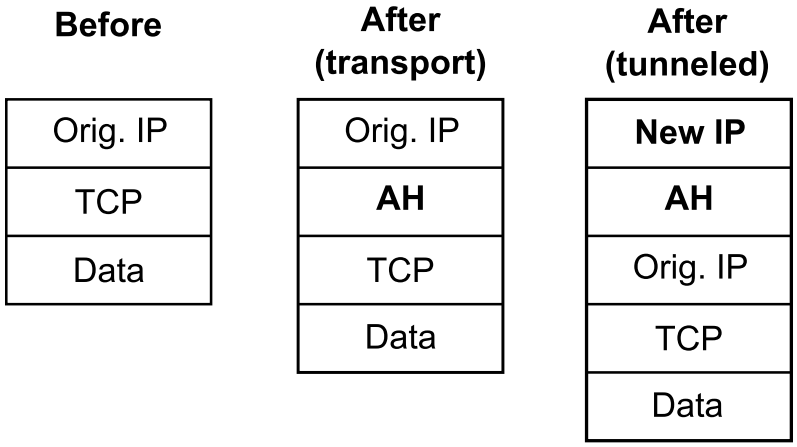

Figure 3.1: Before and after illustrations of the IPsec Authentication Header being applied in either transport or tunneled mode (after [66])

AH, as shown in Figure 3.2, carries a field known as the integrity check value (ICV) that is a variable-length field used to hold an integrity check over certain fields from the IP header, the AH itself, and the IP payload, which contains transport and application layer content. In order to provide protection coverage for the IP header while not being disrupted by typical and expected changes to fields such as TTL decrements, the AH specification defines each IP field as either *mutable* or *immutable*. The mutable fields are DSCP, ECN, flags, fragment offset, TTL, and the checksum. The immutable fields are everything else: version, Internet header length (IHL), total length, IP identification (IPID), protocol, and IP addresses. All mutable fields are zeroed out in order to compute the integrity check.

Unfortunately, ease of traversal and key exchange can be debilitating issues when dealing with IPsec. Regarding traversal, IPsec acts as an additional layer on top of IP (it even has its own protocol numbers) so all systems along a path must be able to properly support it. Also, a number of complex and subtle issues arise when trying to traverse increasingly common NAT gateways [67]. Key exchange issues largely prevent IPsec from being securely applied to general Internet paths (e.g., between a user at a coffee shop and each of the servers of websites they may choose to visit). Anonymous clients of a website would have to be able

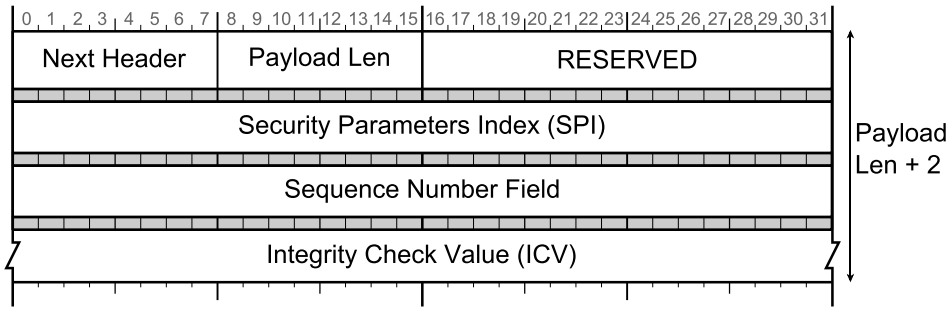

Figure 3.2: Fields that define the IPsec Authentication Header (after [66])

to coordinate in advance to share the same key or trust the same set of certification key signing authorities as the server and to date no such public key infrastructure (PKI) has been widely standardized.

TCP MD5 Signature Option

Developed as a more end-to-end alternative to IPsec, the TCP MD5 Signature option was standardized in 1998 [68]. The designers wanted a more lightweight solution that would not suffer from the same traversal issues as IPsec, but would still be effective at preventing the arbitrary injection of packets (e.g., TCP RSTs) within a stream. To accomplish this protection, a 16-byte message authentication codes (MACs) is calculated using the MD5 hashing algorithm [69] and placed in the options space of every TCP packet of a connection, enabling packet authentication and integrity.

The primary motivation behind this new option was the noticeable increase in IP spoofing attacks in the mid-1990s [70]. In particular, long-lived flows whose sequence numbers grew beyond the allocated 32-bit space and wrapped back around were at great risk since it made guessing a valid number much easier. Border Gateway Protocol (BGP) connections between BGP peering routers are one such scenario with TCP MD5 being the recommended solution [68, 71]. TCP MD5 works for BGP peering routers since the requirement to manually install pre-shared keys can be coupled with the process of establishing trust relationships between routers.

TCP Authentication Option

The TCP MD5 standard later evolved into, and was obsoleted by, the more generalized TCP-AO described in RFC 5925 [72]. TCP-AO is a new TCP option that enhances the strength and flexibility of the MACs over that of TCP MD5 by allowing for the use of stronger algorithms (e.g., AES and SHA [73]). However, TCP-AO still requires that keys and certain session parameters, such as which MAC algorithm to use and whether TCP options are covered, be established manually or by an out-of-band mechanism. It is primarily used for the same purpose as TCP MD5—long-lived BGP connections.

SSL and TLS

SSL, and later TLS, use public key cryptography to authenticate one application to another and establish a session key for data encryption [74]. It can be used to encrypt traffic from a variety of applications, but each program that desires encryption must be modified to use it. Since SSL operates above TCP, it only protects application layer messages and not IP and TCP packet header fields, meaning that it cannot be used to protect packet headers from middlebox modifications.

Summary

Tamper prevention schemes that require trusted keys for authentication tend to be cumbersome and require some type of advanced coordination between servers and clients (i.e., key distribution). This requirement makes these mechanisms difficult to use as a general integrity mechanism that can protect open services with anonymous clients. If the protected key distribution requirements are relaxed, encryption can be more easily engaged as seen in the following section.

3.1.2 Opportunistic encryption

Difficulties with key distribution and infrastructure have led to various opportunistic approaches to encryption. With opportunistic encryption, the authentication requirements are weakened meaning that establishing an encrypted session is easy, but no guarantee is made that the session is to the correct party vice some MITM. Figure 3.3 illustrates an example of a middlebox, Mallory, fooling two endpoints, Alice and Bob, into thinking they have just exchanged keys with each other. Instead, Alice and Bob each performed a separate key exchange with Mallory, who now proxies data between the two connections. Note that

Mallory now has read and write access to Alice and Bob's otherwise encrypted communications. We discuss two opportunistic encryption approaches that successfully protect information from passive but not active adversaries: IPsec Better-than-nothing-security (BTNS) and Tcpcrypt.

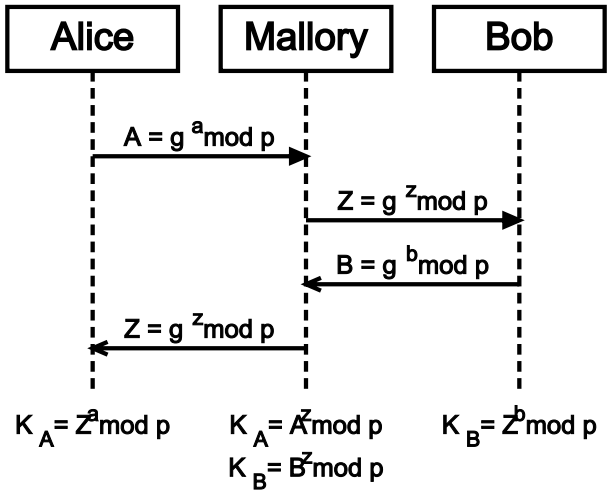

Figure 3.3: Example of a Diffie-Hellman key exchange between two endpoints, Alice and Bob. A third system, Mallory, acts as the MITM and intercepts their traffic (from [75]).

IPsec Better-than-nothing-security

BTNS is an unauthenticated mode of IPsec [76]. BTNS uses self-signed keys to avoid the step of having to verify identities. Due to its easy vulnerability to MITM attacks, the authors recommend combining it with a higher-level authentication mechanism that cooperates with IPsec. An advantage of BTNS is that it is effective against off-path attacks where the adversary does not hold a MITM position, but instead uses spoofed IP addresses to attempt to inject packets in the communications. A disadvantage is that BTNS still comes with all of the traversal issues inherent in IPsec.

Ultimately, even if IPsec could be deployed between anonymous clients using BTNS, it would still have trouble protecting all packet headers in every situation. For instance, ESP in transport mode does not protect IP headers at all. In tunneled mode, the IP header that is encapsulated is protected, but the outer IP header is not. Further, if ESP gets unwrapped at a gateway—as is common for remote teleworkers using corporate virtual private network (VPN) clients—all of the packet headers traverse the remainder of the path unprotected.

ESP in tunneled mode is widely assumed to be the most method of IPsec operation [77], but no measurement studies were found to confirm this. The claim is sensible, however, since tunneled-mode ESP has the least issues with traversing NATs, leaving most VPN solutions to use that approach.

Tcpcrypt

Langley proposes widespread encryption of Internet traffic in order to limit trivial eavesdropping on public networks and click stream monitoring by unfriendly ISPs [78]. The suggested strategy is to fit a Diffie-Hellman exchange into the TCP options space to bootstrap a session key. The work did not provide an implementation of this suggestion, but the idea evolved into the extension Tcpcrypt, which has hashed out the particulars behind putting a key exchange into the TCP options space and has an actively maintained implemented.

Tcpcrypt is an extension to TCP that was released in 2010 to perform opportunistic encryption of TCP connections [24]. The protocol defines new key exchange primitives called `CRYPT` options for the TCP option space that enable encryption keys to be negotiated directly within TCP. Once a shared session key is negotiated, TCP enters the `ENCRYPTING` state where all TCP payload data is encrypted. Also in the `ENCRYPTING` state, all segments include a `MAC` TCP option that authenticates the ciphertext payload along with most fields from the TCP header. Port numbers are specifically omitted and relative offsets from the ISNs are used instead of the nominal values. These fields are left unauthenticated since middleboxes commonly change them and would otherwise cause a packet to fail authentication and be discarded.

3.1.3 Summary of Tamper Prevention Limitations

Many of these protocols that provide strong security guarantees share a common theme: strategies that require the network to understand a new protocol or extension exhibit interoperability issues [19]. Tcpcrypt, at least, is incrementally deployable due to its use of the options space. If an endpoint does not support Tcpcrypt, it simply ignores the option. A problem with this approach, however, is that options may be dropped or mishandled by any system in-line. Unfortunately, paths that would mishandle the Tcpcrypt option are precisely the same paths where TCP needs information about packet header modifications the

most. In reality, Tcpcrypt is another in a long line of extensions that need to be checked for correctness rather than it being a viable solution to the problem.

Tamper prevention solutions are also uncooperative with middleboxes that make desirable changes to packet headers (e.g., IPsec and NAT [67]). Network administrators must still be able to enforce their corporate policies and, as a result, a solution that leaves packet headers unencrypted and adjustable when needed would gain much wider acceptance. Implementing a solution that simply adds integrity information to packets rather than complete header encryption would allow for a higher level of interoperability and acceptance. We maintain that interoperability with current network devices could be achieved while still being able to successfully detect modifications and improve performance in the presence of disruptive middleboxes.

3.2 Avoidance

Recently, the research community has paid significant attention to various means of explicitly accommodating middleboxes and thoughtful redesigns of middlebox architectures. The community is well-aware of network administrators' increasing reliance on middleboxes in their networks [79], a market estimated to reach more than $10 billion by 2016 [80]. This figure alone is evidence that middleboxes are here to stay, and of the value that they provide to networks and their customers.

An early proposal by Walfish *et al.* from 2004 introduced a new architecture that gives all entities globally unique identifiers in a flat namespace while allowing for explicit intermediate packet processing [81]. The idea behind this approach is that sender and receiver endpoints explicitly delegate their packets to be subject to a middlebox's services, such as address translation by a NAT device or filtering for security by a firewall. One major drawback to the approach is that it assumes a cooperative middlebox delopyement; no mitigation is made to stop in-line middleboxes from violating transport and application layer packet contents. Ultimately, this ambitious proposal was never used by the community, nor would it have truly addressed the full range of broken middlebox issues end-to-end which are discussed further in Section 3.2.3.

In the time since, vendors of wide area network (WAN) optimizers have also recognized the problem of middlebox cooperation in traffic modifications and have begun adding their

own TCP options. For example, the IETF is currently drafting a new TCP Middlebox Option, which requests voluntary detection of other middleboxes along a path [82]. The option only helps specific devices from certain vendors that support it; legacy devices will not only not support it but will likely strip it as well. The option also has no end-to-end meaning and is commonly removed from a packet before it reaches its destination.

3.2.1 Software-defined middleboxes

Beginning in 2011, many in the community began to advocate for the application of principles from SDN to middlebox architectures [22, 83]. SDN is a new and emerging network architectural design strategy that decouples the intelligence and traffic processing logic from the physical forwarding components in network devices [84]. The goal of SDN is to be able to centralize all of the decoupled intelligence components, making them more easily managed and enabling new open and standardized interfaces to the network control plane. The advantage of applying SDN to middleboxes is that the centralization can reduce the sprawl of standalone, non-cohesive middleboxes and unify control over middlebox operations. Since then, a variety of solutions employing these principles have been developed [11, 85–88].

xOMB (pronounced "zombie") [85] is a modular software-defined middlebox architecture that utilizes commodity hardware and operating systems to implement a middleboxes services framework. While debugging the modules is easier than a standalone middlebox, the framework does not implement any checks for packet modification correctness, so having correctly operating and up-to-date xOMB middleboxes still depends on the skill and attention of the local network administrators.

CoMb [87] is a top-down redesign of middlebox infrastructure that seeks to develop a more open and extensible middlebox platform that will allow for the consolidation of the middleboxes on a network, reducing device sprawl. With a minimal performance overhead, CoMb reduces the number of different devices and different platforms by consolidating middlebox functionality within a single logical controller that can be more centrally managed. A prototype built using the Click modular router [89] showed benefits to the cost of provisioning a new middlebox and reducing the maximum load across the network as the middlebox deployment is adjusted to changing traffic workloads.

Software-defIned Middlebox PoLicy Enforcement (SIMPLE) [88] is an effort to restructure middlebox processing within the network. Designed to work within the constraints of pre-existing middleboxes and SDN interfaces, SIMPLE requires no changes to a network's current middlebox deployment—only configuration of SDN-enabled switches is required. SIMPLE uses a controller made up of three key modules (ResMgr, DynHandler, and RuleGen) to apply a high-level middlebox policy to a network. Middleboxes are treated as non-adversarial blackboxes and rules for their input-output behaviors are automatically learned by the controller. The authors achieve close to 95 percent accuracy in matching original middlebox capability using their protocol-agnostic approach. SIMPLE's primary benefits are to deployment flexibility and load-balancing efforts. For example, they were able to achieve the same maximum load benefits as CoMB without having to modify or consolidate the middleboxes.

3.2.2 Outsourcing middleboxes

Jingling [86] is a prototype outsourcing architecture where the network forwards data out to external "Feature Providers" that can dynamically adjust to changing traffic loads. The feature providers apply equivalent middlebox functionality to the network's traffic so that the network can eliminate their own local middleboxes, thereby reducing cost and management complexity. This technique allows consolidation of middleboxes from multiple networks under one authority that can, theoretically, do a better job of configuring and updating the middlebox deployment. The relation to our problem is that Jingling could help proactively address broken and inadvertent middlebox behaviors, depending on the administrative dedication of the network's operator.

APLOMB [11] is a service to outsource certain types of middlebox processing to the cloud for ease of management. An APLOMB gateway device is installed so that it is logically co-located with an enterprise's gateway router and replaces all of that enterprise's middleboxes. The APLOMB gateway securely tunnels all applicable traffic out to a selected datacenter cloud presence where the middlebox processing is applied to the traffic. An effort is made to reduce the latency and bandwidth inflation penalties involved while still achieving the equivalent functionality of a traditional middlebox.

3.2.3 Summary of limitations with SDN approaches

While the schemes presented in this section make it easier to manage middlebox deployments and keep them up-to-date, they depend on deployment and use. The authors of SIMPLE note in their review of prior work that most SDN-based middlebox schemes exhibit significant barriers to adoption and lack the necessary incentives for a network operator to overhaul their entire middlebox deployment. Lack of deployment inertia is the reason the authors tailored the design of SIMPLE to work within the constraints of legacy middleboxes and existing SDN interfaces—to reduce the impact of adoption and make for easier deployment.

Even if wider deployment is achieved, there is still no guarantee with any of these schemes that the problems mentioned in Section 2.2 would be fully eliminated. Each of these schemes only makes debugging easier by consolidating middleboxes where they can be more easily managed than traditional standalone middleboxes with closed interfaces. There is also no way to expose to TCP within the end-to-end environment that it operates how it should adapt to broken middlebox behaviors in the instances which they still occur. Since the frameworks themselves do not implement any validation for protocol correctness on packet modifications, misconfigurations and non-standard behaviors will still be possible.

There is an even more fundamental inhibitor to the efficacy of these schemes in solving the broken middlebox problem: incentives and (lack of) policy. All of these software-defined management approaches are confined to single administrative domains—domains which may or may not have the incentive or policy to convert its middlebox deployment to a SDN-based approach. Also, the Internet is made up of a large number of these separate administrative domains; a reasonable analog is somewhere between the number of autonomous systems making IPv4 and IPv6 address announcements (over 56 thousand as of May 2014 [90]) and the number of domains observed in DNS (over 168 million as of Jan 2014 [90]).

Due to the fractured nature of the Internet, TCPs in the wild must still contend with a wide variety of middleboxes, both the advanced SDN-based ones and the legacy standalone devices. Furthermore, even if many subnetworks began to adopt and implement one or more of these SDN schemes, we would still be left with a fractured control environment. The

entire Internet would have to go to a single logical controller to truly ensure that misconfiguration and legacy issues could always be addressed. Last, there is the currently unresolved issue of how to implement a validity checker on top of these modernized software middleboxes. No matter the ultimate trajectory of the current SDN middlebox movement, there is still great value to be had in an end-to-end solution that will work over all paths, particularly ones with remaining pockets of legacy middlebox deployments.

3.3 Detection

The networking community has paid a great deal of attention to detecting modifications to packets, but traditionally the assumed cause of modification has been simple transmission errors. Only recently have middleboxes been considered a source of packet modification.

3.3.1 Simple checksums

Protection against transmission errors was considered during the design of the Internet protocol suite, and is built into the stack via various link-layer mechanisms and network and transport layer checksums.

Two of the most commonly used link-layer protocols both employ a cyclic redundancy check (CRC) to detect corrupted frames: Institute of Electrical and Electronics Engineers (IEEE) 802.3 (Ethernet) [91] and IEEE 802.11 (WiFi) [92]. Also, not only does the 802.11 protocol family include a check sequence in each frame, but ACK frames are used to confirm a receiver's proper receipt of a frame. The absence of one after a certain period of time is the signal for the other end to retransmit [92]. The extra precautions are taken with WiFi due to its inherently higher error and loss rates. Ultimately, these checks can detect modifications on a single link, but they have no end-to-end significance.

The IP, TCP, and UDP protocols all include a checksum as well that is 16-bits in length. The Internet checksum algorithm that is used is weaker than a CRC, but can be efficiently implemented with very fast binary operations. The algorithm computes a one's complement sum of 16-bit chunks of data that are to be included in the checksum [9]. The IP checksum only covers the IP header, while the TCP and UDP checksums cover a pseudo-header that includes some IP header fields, the TCP/UDP header fields, and the packet data.

The IP checksum does not apply well to the solution we seek because it does not cover

any transport layer fields. Furthermore, IP checksums are not end-to-end as they cover fields that change in transit such as the TTL field. This fact forces the checksum to be rewritten at each hop, meaning that errors occurring within those intermediate systems, the middleboxes that introduce the errors, will not be detected. Even if the IP checksum could be changed so that it does not cover any mutable fields, it would still not work because correct checksums are required for packet acceptance and so all middleboxes must, by necessity, recompute the checksum even if they change an immutable field. Another point to note is that IPv6, the anticipated eventual replacement for IPv4, does not even include a checksum.

Transport-layer checksums do have end-to-end significance, but they must be overwritten any time a middlebox needs to modify a transport layer field. Since a correct checksum is required for an endpoint to accept a segment, middleboxes must recompute the checksum anytime they make a change. There is then no way for either endpoint to know whether the checksum received is the same as the original checksum. There is also no way for a sender to know if the received checksum was even correct, let alone the same as the original.

Stone and Partridge note this deficiency of a feedback mechanism and suggest the addition of a new ICMP parameter to alert the sender of a failed checksum [93]. A problem with this out-of-band method is the reliance on the availability of a secondary communications channel, ICMP, which is commonly blocked by middleboxes as noted in Section 2.2. Relying on ICMP may therefore inhibit the ability to communicate integrity, especially on those networks and paths most likely to modify packets and fail integrity. Furthermore, should a new ICMP type be defined to carry feedback of a failed checksum, it could take a long time before networks begin to permit it (e.g., some networks filter different types of ICMP messages at the firewall).

Another problem with checksums in general is the lack of granularity down to individual header fields. All of the checksums discussed in this section only provide a binary answer as to whether the header as a whole has been modified. A change in any single field will cause the whole checksum to fail to match, which will not give a TCP the full information it needs to reason about the correctness of a path.

3.3.2 Application layer approaches

Several approaches also exist at the application layer: Switzerland [94] by the Electronic Frontier Foundation (EFF), Netalyzr [12] by the International Computer Science Institute (ICSI), and the "Echo Mode" in `nping` by the makers of the popular network reconnaissance tool nmap [95]. All of these tools have some ability to check if packets are being altered by middleboxes.

Switzerland was primarily developed as a network neutrality analysis tool to detect when internet service providers (ISPs) were interfering with traffic. The tool is marketed by the EFF as software to "test your ISP." Switzerland works by having a pair of communicating end hosts run software that compares packets sent against packets received at the other end by cataloging mini-hashes on a third-party server. This approach can be used to detect in-network modifications to network packets, and even ones that have been completely forged and injected by an ISP. Unfortunately, the software is not very widely used, and usually only installed when users notice something wrong. It is not integrated into TCP, both parties must be running the Switzerland client, and the service requires availability of a third-party server.

Netalyzr is a Java-based applet that performs a multitude of checks between the Java client and a set of back-end servers to aid in network diagnostics. The tool can only spot traffic modifications if they occur on paths between the client host and one of the maintainers' back-end servers. Another limitation is that the Java security restrictions severely limit the lower-level networking tasks that can be performed. For example, the maintainers of the tool cannot view sent or received TCP sequence numbers at the client within the limits of their implementation. As a result, it is largely used to test a connection for more high-level types of ISP interference such as for the injection of forged web content.

The `nping` tool is a much more flexible and powerful version of the typical ping tool found in most operating systems. It has the ability to send packets of many different protocols and offers the user a variety of customization options. `nping` was originally designed to revive the venerable but no longer maintained packet generation and analysis tool hping [96], which has not been updated since 2005. One of the many enhancements added in `nping` is its Echo Mode.

Echo Mode functions similarly to Switzerland, but without the third party server. Two cooperating endpoints run the `nping` software in "Echo Server" and "Echo Client" modes, respectively. When the two endpoints want to test a path, the client connects to the server and performs an application-layer handshake in its own Echo Protocol to let the server know it needs to start listening for the packets that are to be examined and echoed. Essentially, a raw socket or other packet capturing device is spawned by the server to capture the next packet from the Echo Client along all of that packet's headers. The packet is echoed back to the client where comparisons and analysis take place. Any number of sends and echoes can then occur until the session is closed.

A key limiting factor to all of these options is that both ends of a connection must be running the program, which leads to low rates of adoption. This also means that only certain paths, such as the one between a host and the tool's servers, can be tested for modifications. We believe that by extending down into the network stack, a TCP-based technique could lead to more pervasive adoption and the ability to test for modifications with any system on the Internet or application running on that system. The provision of such automatic and continuous debugging information would also have a large impact on the Internet measurement community, as any application running on any Internet host could be considered a cooperating endpoint for testing.

3.3.3 Tracebox

Tracebox [39] is a tool that can detect in-path packet header modifications under certain conditions. It is described as an extension to `traceroute` [37, 97] that works by sending TTL-limited TCP probes and examining ICMP quotations from the ICMP TTL-exceeded messages that come back when the packet expires.

A simplified example of a Tracebox probing session is shown in Figure 3.4. The source system executes the probes by sending packets with increasingly larger TTL values, starting from a TTL of one hop. If the packet has passed through any middleboxes that have made modifications to that packet's headers, it will show up in the quote within the ICMP TTL-exceeded message. This situation occurs in our example when the packet transits through the middlebox along the path between hops B and C. Now, when the source receives the TTL-exceeded message from Router C, the quote will reveal that a middlebox is altering

the IPID values from 123 to 456.

A critical issue omitted by this example is the differences in which portions of the original packet are quoted by a router when that packet's TTL expires. The RFC that defines the ICMP protocol says that TTL-exceeded messages must include a quote of the IP header and the first eight bytes of the IP payload of the packet when it expired [36] (28 bytes in total, assuming no IP options). The purpose behind quoting the first eight bytes is so that ICMP messages could be matched back to their applicable process on an endpoint. A side effect is that it allows the sender of the TTL-limited packet to obtain feedback on the state of that packet's IP headers along a path. For TCP segments, however, these routers would only give visibility into the ports and sequence number since only the first eight bytes of the IP payload are quoted. This restriction makes it impossible to spot changes to other fields of the TCP header, and in particular the TCP options space (which is important because of our desire to protect the extensibility of TCP).

The creators of Tracebox noticed that RFC 1812 [98] recommends a new quoting behavior for ICMP TTL-exceeded messages. The new recommended behavior is to quote as much of the expired IP packet as possible back to the sender. Routers adhering to the newer RFC allow the sender to observe the full packet headers of each TTL-exceeded message and find differences from how the packet looked at origination. The authors found that many newer routers actually implement this RFC 1812 behavior and wrote Tracebox as a means of automating the probing and differencing of the IP and TCP headers.

The benefits of the methodology used by Tracebox are that information is learned not only about what fields were changed, but also the new values of the fields and where along the path the change occurred. Determining which hop along a path is responsible for a modification is going to be difficult for any purely end-to-end-centric strategy. There are also fewer restrictions on which types of packets can be checked; use of the tool is not limited to SYN packets as any packet can have its TTL artificially lowered by the sender. The benefit of Tracebox is likely to increase as so-called "full-quote" routers (i.e., routers that follow the quoting behavior recommended by RFC 1812) are becoming more and more common on the Internet. In May 2005, Malone and Luckie measured ICMP quote lengths from over 84,000 webservers and found that almost 11 percent of quoters returned the full IP packet [99]. In April 2013, the Tracebox authors measured paths to the Alexa top 5,000

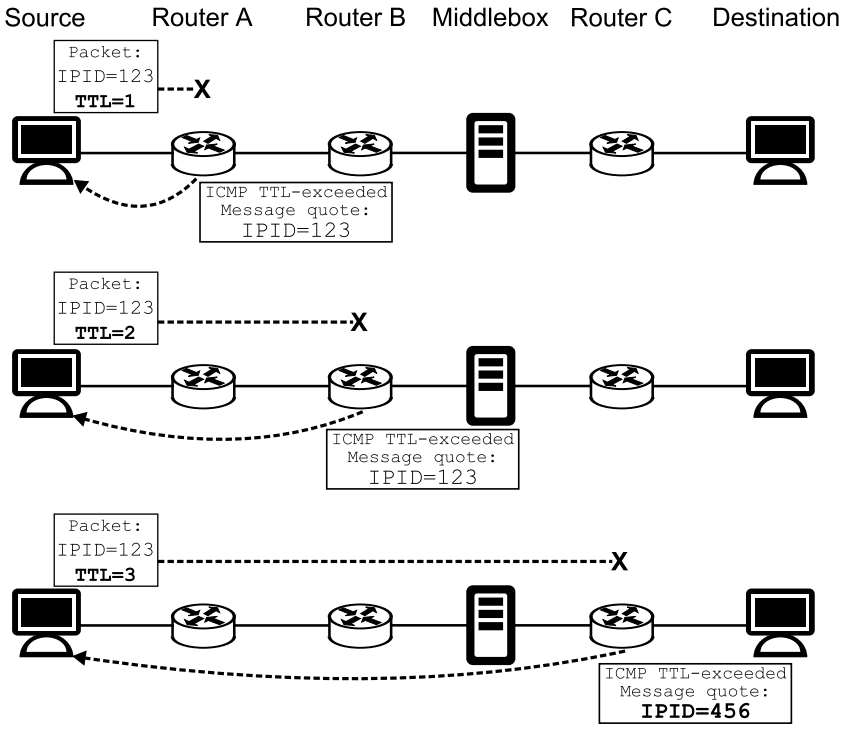

Figure 3.4: Detecting modifications with Tracebox

websites from 72 PlanetLab nodes and found that 80 percent of the 360,000 paths contained at least one full-quote router.

While Tracebox is useful, it does have certain limitations. For one, Tracebox assumes a completely cooperative environment:

- routers have to be trusted to properly quote the packet
- middleboxes and routers must be trusted to not tamper with quotes in other ICMP TTL-exceeded messages being forwarded through them
- policy restrictions have to allow the ICMP messages to make it back to the sender

As seen with PMTUD, neither an open policy for ICMP nor even being properly configured to forward ICMP can be taken for granted [43, 44]. Furthermore, if a client is stuck using an unfriendly ISP, it is trivial for that provider to limit the effectiveness of the technique or induce false readings. The quotes are even subject to misconfigurations themselves [99, 100]. Other issues include:

- lack of visibility to changes occurring in the penultimate hop (because the final hop does not expire and quote the packet)
- a reduction in location accuracy as fewer hops along the path provide full-length quotes
- only forward path modifications are revealed, yet TCP is bidirectional
- requires hop-by-hop iterative querying to most closely approximate full end-to-end information

3.3.4 Summary of Limitations

A common issue among detection-based techniques is that they do not work with TCP so that it would be able to dynamically adapt to middlebox behaviors. Checksums are recognized by TCP, but do not provide the type of information we need. Application-layer or out-of-band solutions are not integrated with TCP and do not have a way to provide the information to TCP. Furthermore, any attempt to integrate them with TCP would only result in a partial solution: the reliance on availability of an application or out-of-band mechanism severely restricts the number of paths for which TCP would have the additional information. In order to be fully cooperative with middleboxes, TCP must have information about changes to packet headers so that it can reason about protocol correctness on its own and adjust its behavior to best match the header modification conditions along a path.

3.4 Advancing the State-of-the-Art

In order to solve our problem as described in Section 1.2, we propose the development of an in-band TCP-based integrity check to detect packet header modifications that occur along a path. The detection solution should endow a pair of endpoints with the ability for each to determine whether their packet headers were modified in transit. Our solution should be incrementally deployable and require no support from transit devices to ensure interoperability.

Through this work, we aim to address an important class of problems due to misconfigured, non-standards conforming, or legacy in-path network elements and endow endpoints with the necessary awareness so they can take some appropriate action, such as disabling an incompatible option or extension.

3.4.1 Security model

In our security model, we operate under the assumption that in-path network elements are not actively malicious. In other words, we shall make no guarantees of protection from strong adversaries and the range of attacks they pose (e.g., man-in-the-middle attacks, cryptanalysis, or side-channel attacks). We also do not strive to provide the endpoints with a means of confidentiality from these devices. Not only would the provision of these guarantees further constrain our solution space, but, as stated in Section 3.1.3, would make our solution less cooperative with properly functioning middleboxes and hurt our likelihood of achieving wide deployment and acceptance within the community.

Therefore, we assume the presence of an inadvertent adversary, a network element somewhere along a path that is not actively malicious but is inadvertently corrupting critical packet semantics. By using this adversarial model, we hope to achieve more desirable interoperability properties in our solution, namely greater flexibility and incremental deployability. As shown in Section 3.1, traits such as these are typically sacrificed when strong cryptography is used. Striking a balance here is often difficult, but we believe our tailored security model will allow our solution to excel in this problem space. Even though our tailored security model does not protect against the powerful list of adversarial middlebox capabilities, we still provide value in the solution space because the issue we are trying to solve is only the detection of inadvertently broken middlebox behaviors (e.g., one's corporate network firewall may be misconfigured but there is no strong reason to treat it as an adversary that performs the actions in the list from Section 3.1 for malicious gain).

3.5 Design Requirements

The preceding survey of the solution space and the current state-of-the-art, reveals many limitations with existing middlebox integrity approaches that we hope to overcome in our design. In order to capture these limitations and highlight our unique approach, we define the following set of design requirements:

- **In-band**: The solution should not require an additional communications channel over the original traffic. Many paths block out-of-band traffic (e.g., ICMP) or treat it differently. By having both the detection and feedback mechanisms in-band, we hope to maximize the detection rate.

- **Minimal overhead**: The design should be efficient and lightweight, resulting in a limited amount of overhead in terms of computation, communication, and RTTs.
- **Symmetric feedback**: It is important that hosts at each end of a connection know whether and how their packets were modified in flight.
- **Incrementally deployable**: The solution should be incrementally deployable and not require updates to in-network elements. The design should also not interfere with end hosts that have not yet been upgraded (i.e., if a connection completes when neither end is using our solution, it should still do so if either or both ends are using our solution).
- **Improves TCP**: The design should endow endpoints with the necessary awareness so that they can take some appropriate action, such as disabling a non-compatible option or extension in order to improve performance. To maximize utility, it should be easily integrated into host protocol stacks.
- **Middlebox cooperative**: The solution should not impede properly functioning middleboxes from making expected and desired changes to packet headers. It should also not be blocked or stop working in the presence of broken middleboxes.
- **End-to-end**: Paths exhibiting modifications are often the same paths most likely to block or strip any new diagnostic functionality. The diagnostic should be properly communicated end-to-end.
- **Granular**: Endpoints should be able to determine which packet header fields were changed.
- **Secure**: The solution should not also enable any new attacks on a system such as amplification, spoofing, or flooding.

One key to developing a *novel solution* that improves upon those currently available is the fresh point-of-view provided by our security model. Much of the prior network research has focused on the edges of the spectrum: protecting integrity from either transmission errors (Section 3.3.1) or from strong adversaries (Section 3.1). When operating under the model of the inadvertent adversary, the solutions developed by those works are either too weak to be useful or make too many sacrifices in pursuit of strong cryptographic assurances. Our approach admits new possible solutions that have the advantage of interoperability with current devices, while still being able to reliably detect common middlebox-induced modifications.

3.6 Comparison with Solution Space

Before detailing our methodology, we place our solution in the context of the integrity and middlebox cooperation schemes described in Sections 3.1–3.3. Table 3.2 informally evaluates the degree to which these relevant prior works meet the corresponding design objectives outlined in Section 3.5. Checksums are featured in the table due to their widespread use within the current protocols. The other three items in the table were chosen as the best exemplar from each of the three groupings under which we categorized the solutions space.

Scheme	**In-band**	**Minimal overhead**	**Symm. feedback**	**Incrm. depl.**	**Impr. TCP**	**Mdl. coop.**	**End-to-end**	**Granular**
Checksums	●	●	○	●	○	●	○	○
Tcpcrypt	●	◔	◑	◕	○	◔	◑	○
Tracebox	○	◔	◑	◕	○	◕	◔	◕
SIMPLE	○	◔	○	◔	◕	●	◔	●
HICCUPS	●	●	●	●	●	●	●	●

Table 3.2: Summary of Related Work

In Table 3.2, Harvey Balls [101] are used to represent the degree to which each integrity or middlebox cooperation scheme meets each of our design criteria. A full ball, ●, means the scheme fully met that criterion. An empty ball, ○, means that the scheme failed to meet that criterion. Other levels imply partial meeting of the criterion with possible caveats. Figure 3.5 presents the same information using graphical overlays. The following lists explain our informal reasoning behind each quantification:

Checksums:

- **In-band**: They are carried within both TCP and IP.
- **Minimal overhead**: The algorithm is lightweight, and only requires extra transmissions when it fails to match.
- **Symmetric feedback**: There is no feedback mechanism.
- **Incrementally deployable**: They are already deployed and boxes understand them.
- **Improves TCP**: Does not help improve TCP under the presence of disruptive packet header modifications.
- **Middlebox cooperative**: Middleboxes can make any changes as long as they recompute the checksum.

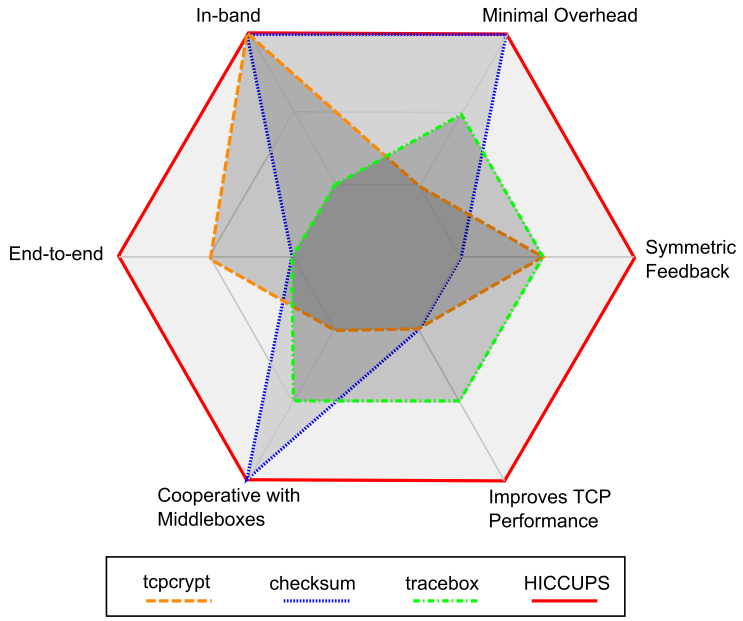

Figure 3.5: Visual representation of properties of related work. This figure depicts the same information as Table 3.2, and is included only to provide an additional means of perspective about the solution space.

- **End-to-end**: Must be overwritten if any packet header modifications are made.
- **Granular**: No granularity to individual fields.

Tcpcrypt:

- **In-band**: Operates fully within TCP.
- **Minimal overhead**: Encrypts opportunistically but uses strong cryptography and uses a large portion of the options space.
- **Symmetric feedback**: Communicates through TCP options, but fails to give status in certain situations.
- **Incrementally deployable**: Hosts that do not understand tcpcrypt just ignore the option, but ossified network architecture must allow transmission of new TCP option type.
- **Improves TCP**: Prevents changes, but cannot help two endpoints optimize parameters for a disruptive path.
- **Middlebox cooperative**: Middleboxes cannot change packet headers once the con-

nection is encrypted.

- **End-to-end**: Will not work on all paths because it is susceptible to having its options stripped.
- **Granular**: No granularity to individual header fields.

Tracebox:

- **In-band**: Relies heavily on ICMP messages.
- **Minimal overhead**: Requires successively fractional RTTs similar to `traceroute`.
- **Symmetric feedback**: Host being probed does not learn any information.
- **Incrementally deployable**: To function, routers need to support RFC 1812-style packet quoting.
- **Improves TCP**: Does not give any information to the TCP stack.
- **Middlebox cooperative**: Middleboxes can still make any changes, but only checks one path.
- **End-to-end**: Cannot determine modifications made by penultimate hop.
- **Granular**: Gives granularity when router quotes full length of headers.

SIMPLE:

- **In-band**: Requires SDN infrastructure along the path in question, which may span multiple providers and networks.
- **Minimal overhead**: Requires testing for correctness of protocol behavior.
- **Symmetric feedback**: Information is retained by the network operator.
- **Incrementally deployable**: All middleboxes along a path must be upgraded in order to realize benefits.
- **Improves TCP**: Correctness-testing in the SIMPLE architecture would help avoid problems.
- **Middlebox cooperative**: Uses dynamic learning module to work with any middleboxes.
- **End-to-end**: Has no effect on middleboxes outside the owner's administrative domain.
- **Granular**: Learns individual field modifications.

TCP HICCUPS:

- **In-band**: Tests TCP and operates completely with TCP.
- **Minimal overhead**: Does not use computationally expensive cryptography.
- **Symmetric feedback**: Each endpoint receives information about the path.
- **Incrementally deployable**: Uses no new options or redefined field semantics that would confuse inflexible middleboxes.
- **Improves TCP**: Can check protocol extensions for correctness and inform TCP.
- **Middlebox cooperative**: Permits middleboxes to continue normal and expected operation, but now endpoint TCP are informed of any packet header changes.
- **End-to-end**: Works on paths that do not modify two of: IPID, ISN, and TCP receive window. Detects any changes that happen anywhere along the path.
- **Granular**: Uses coverage sets to learn individual field modifications.

In the context of these existing and proposed integrity and middlebox cooperation schemes, we endeavor to demonstrate in the following chapters that our design meets our design criteria and represents a unique point in the design space.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Methods for transmitting integrity

Strive not to be a success, but rather to be of value.

Albert Einstein

In order to endow endpoints with the power to detect in-path modifications to their traffic, we must first address the fundamental design decision of how to communicate an integrity check of the packet header fields and any related status information. This chapter examines various methods for transmitting integrity information and analyzes the implications involved with using each method. After targeting in-band TCP for our integrity transmissions, we examine issues and precedent for using various TCP header fields.

4.1 Integrity Properties

To implement an integrity check over the TCP and IP packet headers, based integrity check, the two systems communicating in a TCP session need to transmit to each other a representation of the packet header states as each side sees them. To achieve symmetric notification, they must then be able to compare the state representation as they see it upon receipt with the representation seen by the sender. A primary issue will be to find which fields can be used to carry the integrity check.

4.2 Protocol Layer Overview

The available methods for transmitting integrity are constrained by the current use of protocols in the TCP/IP suite, per our interoperability design requirement. Based on our examination of related work in Chapter 3, protocols from the network layer or above could potentially be used to address detection of in-path packet header modifications. This leaves our range of practical choices with one, or some combination, of: IP, ICMP, UDP, TCP, or an application layer protocol.

Figure 4.1 illustrates the relationships in choosing between each of the protocols to use for integrity transmission. As we go higher up the stack, we achieve better end-to-end prop-

erties (i.e., less interference from middleboxes), but we also decrease in breadth and scope of what types of modifications we can detect. For example, using a separate application protocol may cause us to miss modifications made just to web traffic on port 80. There is also the risk of availability of feedback when any out-of-band protocols are used. In other words, if the modification we would like to detect happen against the TCP header, relying on another protocol (e.g., ICMP) for feedback introduces another variable and another communications channel that may not always be available. More details are included in the sections that follow.

4.3 Application layer

An application layer methodology is the least restrictive and least intrusive to implement and use. Since any application protocol can transit a TCP/IP network[1], we are not limited by the size restrictions of operating within the headers of a pre-defined protocol such as TCP. Not being limited in data capacity could allow the use cryptographically-strong hashing functions and echo entire packets while encrypting them for added protection to facilitate easy and reliable comparison. This methodology is what is currently done by `nping` and Switzerland as described in Section 3.3.2.

There are many opportunities at the application-layer to take advantage of these freedoms and create a new and ideal design, but a critical impediment is the ability to test what we want to test and the low rates of adoption. Each of these issues is easily seen in the discussion of `nping`, Switzerland, and Netalyzr as covered in Section 3.3.2. Ultimately, we desire a technique that can become a natural extension to one of the lower-layer protocols and gain widespread use. Furthermore, we do not want to confine the benefits of inferring path knowledge to a particular application, but rather make them available to all applications. Problematic middleboxes can be found on a multitude of paths within the Internet and in order to properly address the issue, users need a solution that works on a path whether or not their destination is running a specific application layer server program.

[1] speaking ideally, of course—in reality there are a number of exceptions with protocol-aware application layer proxies or gateways [12, 102]

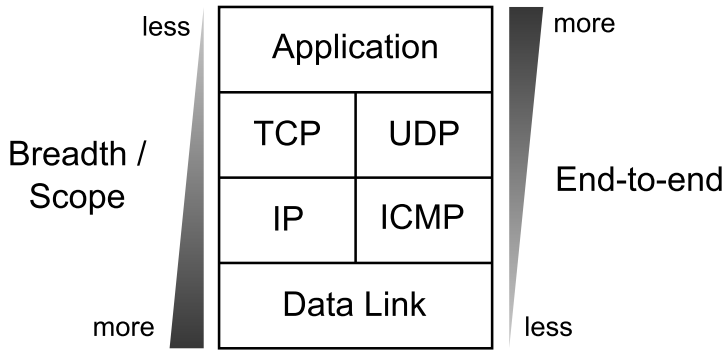

Figure 4.1: Trade-offs to balance when choosing a protocol for our design

4.4 ICMP

Upon initial observation, ICMP is a natural vehicle for the integrity information we need to communicate. The protocol itself is designed to facilitate diagnostics and transmit error and control information about IP packets [36]. In giving initial consideration to using new types of ICMP packets to carry integrity, we generated the following scheme ideas:

ICMP trigger

When a sender is concerned about tampering by a middlebox, the system sends an ICMP message to the destination host to trigger an echo. After receipt of the trigger, the destination echoes back the next packet that it receives from that source host. When the sender receives the echoed packet, a one-to-one comparison can be performed between the two versions of the packet. Since this scheme requires the receiver to maintain state, a thorough security analysis would be required before choosing this method.

ICMP error

As an extension to one of the pre-existing tamper detection solutions from Section 3.3, add a feedback component that echoes any packets back to the sender that fail integrity checks.

ICMP hybrid 1

The same as the "ICMP error" method, but only echo the failed packet if an ICMP trigger message has been received from the source. This scheme would likely cause confusion between the cases where the trigger was never received or a packet was just not modified.

ICMP hybrid 2

The same as "ICMP error", but only include the byte offset where the error occurred in the original ICMP error notice. Then, listen for a trigger message from the source to see if that system wants the full packet echoed back.

An issue shared by all of these approaches is that they all require the availability of an out-of-band mechanism. Given the issues with PMTUD previously discussed in Section 2.1.2, we would expect to encounter a sizable portion of paths where our ICMP messages would be blocked. Therefore, the number of paths that support detection with ICMP would be a subset of the in-transport results. Since our targets for detection are paths that experience issues due to misconfigured and non-standard middleboxes, it is disadvantageous to rely on ICMP since the likelihood of missing results on these problematic paths is increased. While most would likely agree with the notion that getting feedback sometimes is better than never getting feedback, the very same times one gets nothing are precisely the times when feedback would be the most useful and desired. As a result, we determine there is no additional benefit to ICMP over an in-band method with respect to having widespread ability to test paths.

4.5 TCP/IP

A benefit of working within TCP and IP is that TCP is also the packet header that contains the vast majority of the fields we would like to examine for in-path modification (e.g., based on our analysis in Chapter 2). Operating within TCP and IP would mean that integrity transmissions could occur in-band. The result is that if a TCP port is open and a connection taking place, we should be able to successfully transmit our integrity bits since we would have access to a readily available communications channel.

After deciding to leverage TCP, the next question is how to do so. As mentioned in Section 2.1, both TCP and IP each contain a number of fixed-length fields and up to 40 bytes in the options block. The fixed-length fields all have currently defined semantics associated with them, with the options space being somewhat more flexible. The options space is a natural field to be used by extensions, but it also has issues that we will expand upon in the following section.

4.6 TCP/IP Options

An examination of previously proposed protocol extensions suggests places to avoid as well as many opportunities. The obvious place to begin looking for space in the headers is the options fields. The consensus here is mixed. While TCP options may be acceptable and commonly used, IP options, as Fonseca *et al.* put it, are "not an option" [18]. This is due to IP options not being well supported in the Internet. Many devices aim to minimize processing time when routing and forwarding IP packets by ignoring or stripping IP options and only examine the first twenty bytes of the header. Even in cases where IP options are processed, they have to be done so on what is commonly referred to as the "slow path."

TCP options, in contrast, tend to have much more flexibility, do not impose a packet forwarding performance penalty, and carry with them fewer traversal issues. Prior measurement studies found that TCP options, even non-standard ones, are often (but not always) maintained by middleboxes during transit—Honda *et al.* found that in the worst case, 80 percent of paths maintained unknown options [19] for both SYN and data packets. Use of the options are common; the two TCP-based security schemes described in Section 3.1 use their own non-standard TCP options for extra space to carry key exchanges and integrity values. With that fact in mind, a couple of alternatives could exist using TCP options.

One alternative could be to allow for the sender of any TCP packet to include an echo trigger within the TCP options of that packet. Upon receipt of a packet with the trigger in the options, an ACK message would be generated for that packet with a quote of as much of the headers as possible in the options of the ACK message. Such a mechanism would allow the sender to perform a comparison of the two versions of the headers, however some type of compression strategy may be needed since there would be more headers than could fit within the remaining space for TCP options.

A second, more practical, alternative would be to create a hash of the packet headers and include that within the options. Each end of the connection could include a hash with each packet they send and perhaps some status information to try to determine specifically which fields were modified. Unfortunately, while this idea sounds both sensible and practical, it encounters the same issue as the ICMP-based methods: instances of broken middlebox behaviors would make feedback unavailable in precisely the same times it is most desired.

Even if the majority of paths maintain unknown options, it still only takes a small fraction of paths that remove or modify the options in order to prevent or demotivate a new protocol from being deployed. In addition to problems with options being stripped, the space for them is becoming overcrowded. Ramaiah takes note of the various proposed extensions competing for TCP option space and finds that the originally designated 40 bytes of options are no longer able to meet current demands [35]. When the various proposals are considered, the options space is already overused while demand continues to grow.

4.7 TCP/IP Fixed-length Fields

Returning to the fixed-length fields, opportunities for transmitting an integrity value exist if it can be placed in a field that is re-purposed from its current use today, or *overloaded* onto a field that has a loose set of constraints on its values. For example, if a field is used to transmit any non-specific number (e.g., IPID), then we can replace it with an integrity value we constructed and no set of rules would exist that middleboxes could use to block our integrity. Re-purposing a field is a delicate task and would require great care to ensure middlebox compatibility. We next examine a variety of specific possible uses within the fixed-length fields.

4.7.1 URG pointer

Some of the workarounds described by Ramaiah [35] suggest more possibilities such as re-purposing of the TCP urgent pointer. The work suggests that use of the urgent pointer field has largely declined, but re-purposing it is likely suboptimal for our needs given that it has prior semantics attached to it, and re-purposing it may cause odd behavior with legacy devices – exactly those we wish to interoperate with cleanly. We note this possibility so that the assumption can be examined in later work.

4.7.2 Offset checksums

Another workaround suggested in the work by Ramaiah [35] is the interesting notion of a type of deliberately incorrect checksum—here termed "offset checksums." The idea is to send multiple segments covering the same sequence number space, with all but one copy having an incorrect checksum by design. The additional segments will be dropped by a traditional TCP due to the incorrect checksums, but could give special meaning to the additional packets in a revised TCP. This solution has nice interoperability properties because

any receiver that does not understand the messages will just think they are corrupt and gracefully drop them. Downsides to this option include the excess bandwidth consumed, and the fact that this type of behavior may likely trigger intrusion detection system (IDS) alerts. This idea also relies on the proper behavior of endpoints and in-network elements with respect to bad checksums—that they will gracefully ignore them instead of dropping them early or worse, fixing the checksums.

4.7.3 TTL

Generally, the uppermost one or two bits from the TTL IP packet header field could reasonably be expected to be maintained across a path traversing the Internet. The field is eight bits long, meaning that a packet sent with the maximum TTL value would be able to transit 255 hops before expiring. This value is often more than enough needed to reach the destination, and router decrements would only likely occur up to the third most significant bit (63 hops). It is unclear though whether these expectations would hold in the future or even in all cases today. The best approach here would be to act conservatively and wait until after a session has been established and we observe the incoming TTL to approximate the path length.

4.7.4 Flow control window

As described in Section 2.1.3, the flow control window should have a specific setting based on system resource availability and an estimate of a path's capacity—the product of end-to-end bandwidth and delay [47]. In general, this field cannot be overloaded but we note that its latest value is updated with every single TCP packet sent in the connection. If we overload the window on a single packet, for instance, the SYN packet, it will be updated with the next ACK or data packet, replacing our overloaded value with the true value desired by the host. As a result, we can consider reusing the initial flow control window field for transmitting some bits.

4.7.5 IPID

Another area of promise are fields that are expected to be able to hold any random number. There are two such fields in the IP and TCP headers: the IPID and the ISNs, respectively. The IPID field is a 16-bit field in the IP header that, in the event of fragmentation, is used to differentiate fragments of one packet from another [31]. It does not matter what the

value is as long as it is unique to each packet when fragmentation occurs. When there is no fragmentation, the value of the field is of no use since reassembly is not required. Despite this, all IP packets, regardless of whether they are fragmented or not, carry an IPID in their fixed-length headers.

The IPID field is particularly safe to re-purpose when another field in the IP header is enabled: the DF flag. The DF flag tells downstream devices along a packet's path that fragmentation of that packet is "NOT permitted" [31]. In other words, when the DF flag is enabled, IPID has no direct purpose and can safely be reused, even if multiple packets have the same identification value. RFC 6864 [103] defines such packets as *atomic*, meaning they have not yet been fragmented and any in-network fragmentation is prohibited. The conditions for an atomic packet are defined in Equation (4.1).

$$(DF = 1) \wedge (MF = 0) \wedge (frag\_offset = 0) \rightarrow atomic \tag{4.1}$$

When a packet has been marked as an atomic packet, RFC 6864 allows for reuse of the IPID field. Specifically, the RFC states that "originating sources MAY set the IPv4 ID field of atomic datagrams to any value" [103]. As an example that IPID reuse is viable, note that the IETF is currently discussing a proposal by Briscoe that would redefine semantics for the IPID field on atomic packets that also have the reserved bit set [34].

We were able to initially verify the notion that fragmentation is uncommon in today's Internet by examining a 30 minute Internet backbone capture provided by the Cooperative Association for Internet Data Analysis (CAIDA) [104]. The capture was taken from direction A of the 10 Gbps equinix-sanjose link on 18 April 2012 from 1300 to 1330 UTC. We found that of the over 900 million packets in the trace, 99.99 percent were IPv4 packets and only 0.14 percent of those were fragments of some kind. These results are encouraging for reuse of the IPID field since such little fragmentation occurred and the vast majority of packets, about 91 percent, were atomic (i.e., the DF flag was set). While our analysis is limited to a small window of time on one link, it is a large Internet vantage point and provides a general idea of what to expect when examining IPID use in greater depth.

4.7.6 Initial sequence numbers

Another field that can be overloaded is the TCP ISN. While sequence numbers are important for reliable stream reassembly, the initial value used in a connection is not critical to the process. As a result, ISNs provide a single opportunity for each endpoint to send 32 bits of information to the other at the time a connection is opened.

However, one area where the ISN value is critical is in how it affects the predictability of the stream. As long as the ISN retains a sufficient amount of unpredictability to prevent spoofing and injection attacks, hosts can choose any number they would like as the first sequence number of a connection.

One final concern is to maintain compatibility with features that currently overload the value of an ISN. The only widely used scheme that does this is SYN cookies [105]. SYN cookies are used to mitigate half-open connection flooding where an attacker sends as many SYN packets as possible, forcing a server to open sockets for the application listening on that port, allocating a block of memory for each. SYN cookies allow the server to encode a cryptographic secret in the ISN it chooses for the SYN/ACK and safely allocate state once the remote TCP completes the 3WHS with the final ACK. A limitation of SYN cookies is that the limited space does not allow for the server to remember many of the options included with the original SYN packet. Due to this limitation, the Linux kernel does not use them for all connections, only resorting to them when it believes it is under a SYN flooding attack [106]. In overloading the ISN, we can maintain compatibility with SYN cookies by simply continuing to employ this logic.

4.7.7 Summary

While overloading and attempting to reuse the fixed-length fields in the TCP and IP headers limits us to a small transmission capacity for our information, they yield many other nice properties:

- good interoperability with middleboxes
- easy testing due to tight integration into TCP
- no additional RTTs or bandwidth consumed
- easy symmetric feedback since the concept of a connection is already well-defined

Table 4.1 summarizes our exploration of the fixed-length fields and presents our findings with respect to how many bits are available for reuse, whether they can be reused throughout the connection, and how many issues we anticipate in using them. We represent the final item qualitatively using Harvey Balls as in Table 3.2.

Field Name	**Total bits**	**Reusable bits**	**Throughout Connection**	**Anticipated reuse safety**
TCP urgent pointer	16	16	Yes	○
IP TTL	8	1–2	Yes	◑
TCP flow control window	16	16	No	●
IP identification	16	16	Yes	◕
TCP initial sequence number	32	32 (less randomness)	No	●

Table 4.1: Summary of fixed-length TCP/IP fields that can support overloading or re-purposing

4.8 Possibilities from Network Steganography

We find it may also be of use to more closely examine the field of network steganography. The goal of network steganography is to locate covert channels within the protocol headers or timing mechanisms to transmit hidden data. Usually these steganographic techniques are developed under the assumption that two end hosts and their network stacks are in collusion and discretely pass data between each other without systems in-between noticing or modifying their data. It is an interesting connection to our work because the goal somewhat parallels our own. We wish to pass the integrity check data through a side channel as well, but do not necessarily bear the requirement that it be covert even though it might be useful to prevent middleboxes from touching it.

An examination of some steganography literature validates our field ideas. Cole uses the IPID field as well as ISNs to pass data [107], Bennet makes use of deliberately erroneous checksums [108], and Luo *et al.* encode hidden data by partially acknowledging smaller pieces of data at a time and having the other end track the amount of bytes acknowledged with each response [109].

While there may be overlap in field usage ideas between our work and that of network steganography, we emphasize that our goals are fundamentally different. We aim to locate fields that will be most compatible with middleboxes. We are not trying to evade detection

or design covert communication channels. Our intent is for the HICCUPS protocols and integrity computation algorithms to be publicly documented.

4.9 Summary

In this chapter, we examined a number of approaches to find a best fit integrity transmission channel for our design. Selection of the right method is critical for meeting the architectural requirements we listed in Section 3.5. Were we to fail to fully consider all implications involved with each choice, we could subject our method to the same limitations as the other approaches in the solution space we describe in Chapter 3.

After surveying the space of possible transmission channels, we design a novel approach to packet integrity and feedback using fixed-length fields from the TCP and IP headers, known as HICCUPS. Our design for HICCUPS is described in-depth in the next chapter. For completeness and documentation for perpetuity, we also include descriptions of other viable methods we designed, but chose not to implement, in Appendix 9.2.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Transmitting Integrity with HICCUPS

Information is the resolution of uncertainty

Claude Shannon

Following the discussion of the transmission methods explored in Chapter 4, this chapter describes our solution to detect broken middlebox behaviors, TCP HICCUPS. Section 5.1 gives the high-level intuition of our design along with forward pointers to the most salient features of HICCUPS. The intuition of HICCUPS is simple: we overload existing packet header fields (Section 5.2) in the TCP 3WHS to encode bidirectional integrity checks (Section 5.3). The integrity checks can cover different header fields, allowing a HICCUPS-enabled host to infer which fields of its packets were modified (Section 5.5) enroute to a particular destination. By intelligently querying for different field sets, the field coverage flexibility enables an efficient process by which complete path knowledge of all packet header modifications can be inferred (Section 5.6).

As a real-world instantiation of our architectural design requirements outlined in Section 3.5, we develop the Handshake-based Integrity Check of Critical Underlying Protocol Semantics (HICCUPS), an enhancement to TCP. HICCUPS can assist a TCP in determining the most appropriate set of end-to-end parameters that best fit the middleboxes present on a given path. In particular, HICCUPS would allow a TCP to reason about how the options and extensions it employs are interpreted by a remote endpoint, and subsequently make inferences about when it is safe to make use of new extensions.

HICCUPS benefits TCP in two primary ways:

1. Equips TCP with the capability to infer the state of end-to-end packet header modifications on a bidirectional path, similar in a sense to how TCP currently infers the end-to-end congestion state of a path. Having full availability of such critical path information would allow TCP to more safely increase the use of performance-enhancing extensions relative to ultra conservative approaches where new extensions

are disabled by default or left to run in "server-mode" *à la* ECN as deployed and configured in modern operating systems[2].
2. Provides early warning of potential middlebox-induced issues with an extension that is enabled by default. TCP could proactively disable or ignore the extension to improve performance over the case where a path is incompatible with the extension.

Our solution helps enable these performance benefits by monitoring the state of packet headers through an in-path integrity exchange, essentially creating a lightweight *tamper-evident* seal across the headers. The results of the exchange allow end hosts to work within the current path conditions to tailor the set of extensions they use to the middleboxes in the path between them. When a broken or misbehaving middlebox is disrupting usage of an option or protocol extension, HICCUPS can not only inform TCP so that it can be disabled on future connections, but can also assist users or network operators in debugging the issue.

HICCUPS benefits network operators in the following ways:

1. Enables a new network debugging tool that can be used to troubleshoot packet header modifications on networks outside their administrative control to any open TCP port on HICCUPS-enabled hosts.
2. Makes it possible for operators to proactively debug network devices and fix subtle configuration issues before having to field related support calls.

5.1 Overview

Working within TCP to enable detection of in-path header modifications while maintaining interoperability with current network infrastructure and end hosts is a difficult systems problem. We first provide an overview of the distinguishing features of HICCUPS:

1. HICCUPS transmits packet header integrity information by *overloading* three header fields of the TCP 3WHS that can contain a flexible value: initial sequence numbers, initial IPIDs, and initial flow control windows. As we showed in Chapter 4, doing so yields the highest degree of interoperability with the widest number of paths, but places tight constraints on the amount of information that can be transmitted. See Section 5.2 for more.

[2]In server-mode ECN, a TCP will not initiate ECN, but will negotiate ECN if initiated by the client.

2. When HICCUPS places integrity information in the TCP sequence number, randomness is added for spoofing protection. See Section 5.2 for more.
3. The integrity information transmitted by HICCUPS includes three 12-bit *hash fragments*, each communicated through one of the overloaded fields in item 5.1. Spreading integrity across multiple fields provides resilience to a single modification affecting any one of the three fields (e.g., sequence number translation). See Section 5.3 for more.
4. Reverse path integrity includes status flags that enable a HICCUPS host initiating an active open to discover when modifications occur to just the forward path, just the reverse path, or to both paths. See Section 5.3 for more.
5. HICCUPS supports field-level granularity in its integrity checks. A set of coverage types allows end hosts to dynamically specify subsets of fields to be protected by HICCUPS. See Section 5.5 for more.
6. As an added protection (e.g., against middleboxes that might, in the future, actively attempt evasion), HICCUPS enables applications to optionally protect the integrity with an *ephemeral secret*. This secret limits false inferences of integrity in the event that a change is made and the integrity is recomputed by a middlebox. See Chapter 6 for more, and for a discussion of how we extend the Linux socket application programming interface (API) to provide this feature in a compatible fashion, see Section 7.2.

5.2 Overloading Header Fields

To minimize interference from legacy and non-standard middleboxes, we *overload* three specific fields in the headers that are allowed a certain degree of flexibility: the TCP initial sequence number (ISN), the initial IP Identification field (IPID), and the initial TCP flow control window (RCVWIN). Each end of the connection chooses its own 32-bit ISN, 16-bit IPID, and 16-bit RCVWIN resulting in a total of 64 bits at each end of the connection to be used by HICCUPS.

As discussed in Section 4.7.6, if we want to add meaning to the ISN, the ISN must remain unpredictable to thwart spoofing and off-path packet injection attacks. We therefore add randomness to our ISN integrity function. The bits of randomness, or *salt*, are sent in the clear to allow the remote host to verify the integrity. We place the random salt value in the

lower half of the ISN and exclusive or (XOR)-encode the integrity information in the upper half of the ISN with the same salt value.

Since the new ISN is created using a function of packet data, it will not be fully random (i.e., the probability of an off-path attacker being able to correctly guess the ISN is greater than 2^{-32}). In the extreme worst case, the probability is 2^{-16}, but that requires an attacker know:

- The flow tuple including the ephemeral port [110]
- The coverage type used (see Section 5.5)
- The exact contents of any packet header fields covered by that type

In practical use, an off-path adversary will not know the coverage type—two of which also cover the ephemeral port.

5.3 Integrity Exchange

Fundamental to HICCUPS is exchanging integrity and communication of the check results. Given a safe and reliable transmission mechanism (Section 5.2), we are able to exchange integrity, coverage, and status. Our objective is to utilize the 64 bits at our disposal in such a way as to be robust against paths that corrupt any of the three integrity exchange fields. In order to withstand a change to any single overloaded field, we place a portion of the integrity information, along with a copy of the coverage or status, in each of the three fields.

Figure 5.1 presents a simplified timing diagram illustrating the exchange of integrity between two HICCUPS-enabled hosts, A and B. Unless otherwise noted, HICCUPS follows the TCP standard and uses standard congestion control algorithms. For example, our Linux implementation retains its default TCP CUBIC behavior, SYN cookie activation threshold, and default use of various TCP options (e.g., SACK, maximum segment size (MSS), window scaling, etc.). Host A initiates the active open with B. Both SYNs of the 3WHS utilize the ISN, IPID, and RCVWIN fields to transmit up to 16 bits each of integrity information, denoted in the figure as A_n and B_n where $n = 1...3$ and represents the ISN, IPID, and RCVWIN, respectively. Note that A_1 and B_1 are encoded with their respective 16-bit random salts.

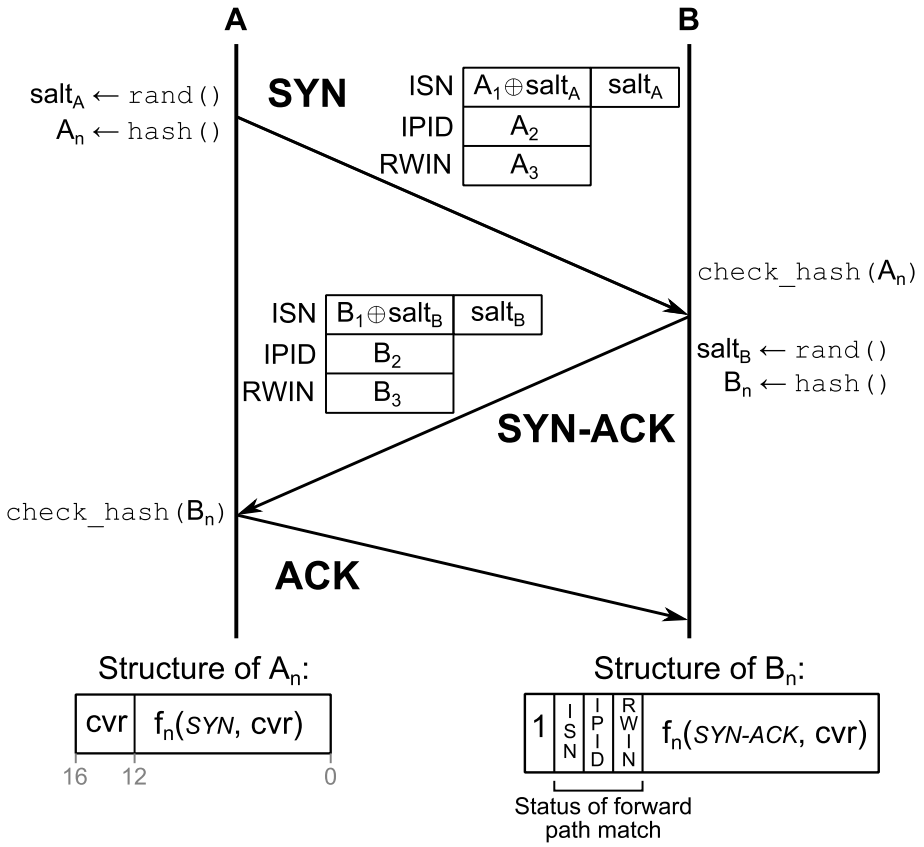

Figure 5.1: HICCUPS integrity exchange: A's `SYN` overloads random fields with integrity and coverage flags. B's `SYN-ACK` encodes reverse path integrity and forward path status.

The internal structure of each 16-bit integrity field A_n and B_n is shown below the timing diagram in Figure 5.1. Integrity values in the forward path from A each contain a 12-bit hash "fragment" and a 4-bit coverage type (*cvr*). The coverage type communicates which portions of the packet header are to be tested. The source populates each A_n with the *same coverage value* to make a best effort that the remote TCP sees a valid coverage type, which reduces the amount of testing it will have to do to verify the hash. Coverage and coverage selection is detailed later in Section 5.5.

Similarly, integrity values sent from B each contain a 12-bit hash fragment over packet header fields in the SYN-ACK, and three bits to return the forward path integrity results to A. A examines these status bits in the received SYN-ACK to infer how its SYN arrived at B. To minimally impact the initial flow control window, the highest order bit of B_3 can be

set to correspond to the true receive window.

In this chapter, we abstract the integrity functions used to compute each 12-bit hash fragment as $f_n(\cdot)$. Thus $f_n(SYN, cvr)$ is the n'th integrity over the *cvr* fields in the SYN packet. The integrity function must be public, allowing the host at the other end of the connection, B, to check the integrity value it receives. Our experimentally validated (see Chapter 7) implementation in Linux uses a combination of truncated CRC32 and Murmur3 [111]. However, HICCUPS could be ultimately standardized to use different functions in the future, based on diffusion and collision resistance requirements.

Tables 5.1 and 5.2 list possible inferences A and B can make during connection establishment. When B receives the SYN from A, it recomputes each A'_n using the SYN header fields as received for each of the specified coverage types. The received integrity A'_n matches the sent integrity if $A'_n = A_n$. If at least two of the three recalculated hashes match the received hashes, B infers that the covered fields in A's packet header were unmodified in transit.

Next, B generates its own (different) salt and integrity values for the return SYN-ACK packet. B's results from verifying each A'_n are echoed back to A by the inclusion of boolean flags for each of ISN, IPID, and RCVWIN in the SYN-ACK integrity B_n. When A receives the SYN-ACK reply from B, it can also check the integrity values. A examines the forward path status bits to determine whether the SYN experienced manipulations.

Using $n = 3$ integrity fields, and a combination of hash functions is crucial given the size limits (12 bits each). HICCUPS infers a packet as HICCUPS-capable when any two integrity values match the locally computed integrity ($A'_n = A_n$). Thus, the probability of a pre-image other than the original generating the same hash with two different hash functions is 2^{-24}, or approximately one in 16 million. While this rate is non-negligible, it is low enough for practical use. Measurement instances requiring higher precision can run a HICCUPS integrity test multiple times to further reduce the probability of a false inference.

5.4 Which fields to protect

In order to determine which fields should be protected by the HICCUPS integrity check, we adapted the list of immutable fields from the RFC for IPsec AH [66]. As noted in Section 3.1.1, the AH specifies the following fields as immutable (i.e., fields that middleboxes

Table 5.1: Possible knowledge gained by host B in performing the integrity check

At B after receiving SYN	**Inference**
$\|\|A'_n = A_n\|\| \geq 2 \; \forall n$	covered SYN fields unmodified
else	SYN modified or A not capable

Table 5.2: Possible knowledge gained by host A in performing the integrity check

At A after SYN-ACK received	**Inference**
$\|\|B'_n = B_n\|\| \geq 2 \; \forall n$	SYN-ACK unmodified
$\sum status_i \geq 2 \; \forall status \in B_n$	SYN unmodified
Both cases above	SYN & SYN-ACK unmodified
else	SYN & SYN-ACK modified; or B not capable

should not modify): version, IHL, total length, IPID, protocol, and IP addresses. Since we utilize the IPID for our integrity transmission channel, we leave that field off the list. We also additionally cover some fields that the AH did not list: the DF flag, the reserved bit, the IP options, and the IP ECN bits.

For the TCP header, none of its fields should be considered mutable (as TCP is end-to-end), but we leave several fields out from our integrity coverages. The sequence numbers and receive window size fields are used to transmit integrity so they are left out. Also, since the checksum will only ever be different when another field was changed (or if a transmission error occurred meaning the packet will be dropped), we leave that field out as well since covering it provides no additional value. Table 5.3 lists all of the fields from the IP and TCP headers that can be protected by HICCUPS.

5.5 What Header Field Was Modified

HICCUPS allows the connection initiator to specify which packet header field or subset of fields the handshake should check. For instance, a HICCUPS-enabled host opening a new connection could choose to only check the TCP MSS option, or it could focus on just the ECN flags. Each individual connection enabled with HICCUPS specifies which fields to check from a pre-defined list. HICCUPS currently supports the 16 coverage types shown in Table 5.4. A type that covers both the IP and TCP options blocks can be used to check other options. Our primary reasoning behind these design choices is directed by

IP Fields Protected	TCP Fields Protected
• version • IHL • ECN codepoint • total length • reserved bit • DF flag • protocol • source IP address • destination IP address • IP options	• source TCP port • destination TCP port • offset • reserved bits • TCP flags • urgent pointer • TCP options

Table 5.3: List of all header fields protected by HICCUPS

the highly constrained amount of space (we require the upper bits of B_n for forward path status) and the initiator being the party that typically chooses which options to negotiate for the connection. The set of coverage types could be extended to explicitly accommodate future extensions if both end hosts are updated with the new list.

The `HFULL` type covers the broadest set of header fields. This type covers all header fields except for those that are expected to change in transit (e.g., TTL) or fields used to carry integrity. The full set of fields covered by this type are shown with a solid gray background in Figure 5.2. The remainder of the coverage types we have implemented are proper subsets of these fields.

In order to check multiple types, a progression of HICCUPS connections can be performed between two endpoints. In this progression, each individual connection uses one of the pre-defined coverage sets. The simplest approach is to check all possible coverages in order. Such an approach would require a separate connection for each, but could be done in parallel to reduce the latency of multiple RTTs waiting for results. Alternatively, the inferences might occur during the natural interaction and multiple connections between hosts. A smarter algorithm that could reduce the total number of connections required is described in Section 5.6.

Selection of a coverage type for a given connection can be done manually by an applica-

Table 5.4: Pre-defined coverage sets

	Coverage Type	**Header fields that are covered**
0	HNONAT	Everything, minus IPs and ports
1	HFULL	Everything
2	HNAT	IPs and ports
3	HNOOPT	HNONAT minus any IP or TCP options
4	HONLYOPT	IP and TCP options
5	HECNIP	ECN IP codepoint
6	HECNTCP	ECE and CWR TCP flags
7	HLEN	Length fields
8	HMSS	TCP MSS option
9	HWINSCL	TCP Window Scaling option
10	HTSTAMP	TCP Timestamp option
11	HMPTCP	TCP Multipath option
12	HEXOPT	An unused TCP option (kind = 99)
13	HFLAGS	IP_DF, non-ECN TCP flags, and TCP SACK_Permitted option
14	HSAFE	Reserved fields, protocol, and version
15	HNULL	Nothing (compatibility check)

tion (Section 7.2) or automatically by the TCP stack. Once a type has been selected, we concatenate the covered packet header fields as input to the HICCUPS integrity functions $f_n(\cdot)$. The only exception is the two bits in the IP header that represent an ECN codepoint. For these two bits, we include their bitwise OR as input. Routers are allowed to modify this field, but only by turning an ECT0,1 codepoint into a CE codepoint. Nothing should set both bits to zero if either one was originally set high by an endpoint (an aberration observed in [20]).

Because a field carrying the integrity, A_n, could be modified, the endpoint analyzing the SYN must test all of the coverage types it sees in the received A'_n. Ideally, none of A_n will have been overwritten, meaning that all three coverage values are the same and only one check must be done. The worst case is that three checks must be done in the event that one or more of A_n were overwritten. If the receiving endpoint finds a match, it *must* use the same coverage type when calculating B_n for the SYN/ACK. Should the receiver fail to find a match (meaning part of the SYN was modified), a majority rule is used on the three coverage types listed in A'_n to determine the coverage to use for B_n. If a majority is not

found, a special coverage type is used in B_n to indicate to host A that at least two of A_n were modified.

5.6 Complete Path Knowledge

Given that only one coverage set from Section 5.5 is used per TCP 3WHS, a pair of TCPs must develop fully granular knowledge of all header modifications over the course of multiple exchanges. When integrity matches for a coverage type that is a superset of other types (e.g., `HFULL`), no further information is gained from additional probing. However, if the integrity fails to match, more specific types can be used next to narrow down the source of the modification.

If integrity using `HNULL` does not match, then either one of two cases is occurring:

1. Two or more of our three integrity fields are being modified.
2. The host with which we are interacting does not understand HICCUPS.

Since `HNULL` is a diagnostic type that does not cover other header fields, it should not fail unless the hash fragments are not present or have been overwritten.

Leveraging this information, we design a path interrogation strategy for HICCUPS. Using HICCUPS to determine the fully granular set of modifications along a path is similar in nature to a search problem. Our informed strategy is shown in Figure 5.3. We begin by checking coverages that are more comprehensive and then narrow the search, eventually checking a smaller sequence of types. Upon our first interaction with a new TCP, we choose the `HNONAT` coverage type since it avoids fields modified by NATs, which are prevalent on the Internet [12]. If we find a match, we conclude the search. Subsequent connection attempts can periodically retest with `HNONAT` in case the path conditions change.

Given that we expect frequent interaction with non-HICCUPS TCP, our strategy employs the `HNULL` type at the next opportunity. By doing so, we can terminate the search in the event that either the other endpoint (due to lack of capability) or middleboxes along the path (due to downgrading the integrity) prevent HICCUPS from being used. The remainder of the strategy searches for header modifications in either the options space or fixed-length fields, iterating through a series of more granular coverage types as needed.

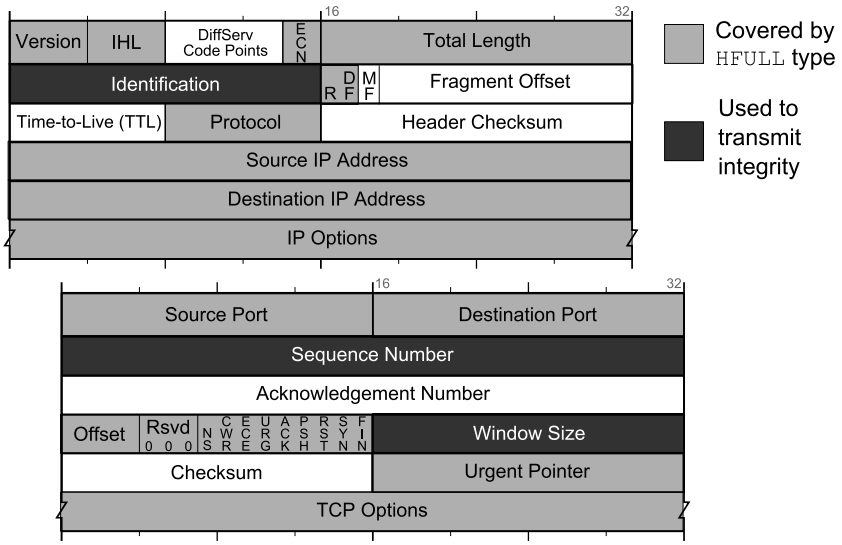

Figure 5.2: Breakdown of IP and TCP header fields in relation to using HICCUPS. The fields in light gray are protected by HICCUPS while the fields in dark gray are used to transmit integrity values.

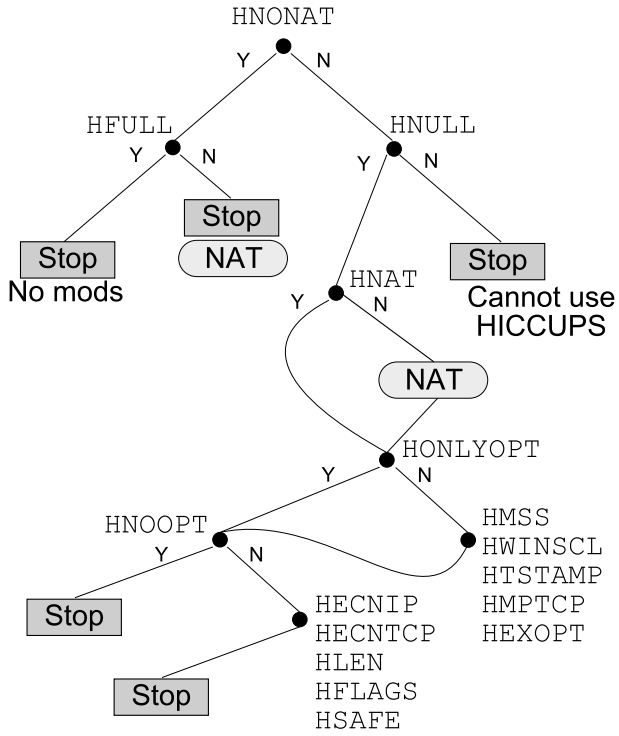

Figure 5.3: HICCUPS Search Strategy

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Protecting Integrity with AppSalt HICCUPS

Good news, everyone!

Prof. Hubert J. Farnsworth

At a high level, HICCUPS is designed to be cooperative with middleboxes. Unlike with checksums, packets will not be rejected by a host due to incorrect HICCUPS integrity. Neither will middleboxes be prevented from making any changes to the packet header. We simply use HICCUPS to inform each TCP of such changes with the intention that TCP will benefit from that knowledge. Our hope is that, by not providing middleboxes with a specific reason to disrupt HICCUPS, that they will not try to alter its integrity values. Further, we expect that most middleboxes will actually support our traffic explicitly when they know about it (e.g., note the work that the IETF has done with the new TCP Middlebox option [82]). However, middlebox administrators have a long history of adopting overly conservative security policies that block simple diagnostic traffic such as ICMP (see Section 2.1.2), so we must take a realistic approach.

This chapter describes an optional enhanced mode of HICCUPS, termed "AppSalt." In our design for AppSalt mode we operate under the assumption that some future middleboxes, when armed with full knowledge of our diagnostic protocol, may willfully tamper with HICCUPS. We therefore wish to increase the effort required of future middleboxes when attempting to circumvent HICCUPS, all while retaining adherence to each of the design requirements of Section 3.5. Ultimately, we created AppSalt to allow applications to optionally add additional protection for their connections, making it more expensive for middleboxes to falsify integrity values.

The strategy described in Chapter 5 is effective at detecting packet header manipulation by devices that are not engineered to evade detection; packet header modifications made by today's middleboxes that are unaware and unable to recognize a connection with HICCUPS integrity will be readily exposed. However, as we previously noted, this may not automat-

ically be the case with all future middleboxes. Engineering HICCUPS to be resistant to strong MITM adversaries, without using authenticity and keyed cryptographic hash functions, is a challenging problem.

While tampering is still possible, AppSalt aims to make undetectable packet header manipulation expensive. Thus, a middlebox must either:

1. Bear the cost of circumventing our protocol
2. Reveal the modifications it makes to the connection endpoints
3. Simply stop meddling in the communication

The value proposition of such a protocol is that item one presents a high enough cost that the middlebox naturally chooses approach items two or three.

A middlebox, M, could disguise a packet header modification by rewriting the integrity values on SYNs from an initiating host, A. Should M also want to modify the SYN-ACK response, it would perform its changes and then recalculate new integrity for the SYN-ACK sent by a responding host, B. Such recalculation of HICCUPS fields could lead to the reduced effectiveness of HICCUPS at detecting potential extension compatibility issues as middleboxes adjust to evade HICCUPS, but then either fail to properly support newer extensions or suffer from a future misconfiguration.

6.1 An Ephemeral Secret

Since our design constraints preclude the use of a stronger construction (e.g., a keyed-HMAC) we cannot outright prevent M from taking a stance similar to that of the middlebox Mallory in Figure 3.3. In that position, a middlebox could split the connection and recalculate valid integrity values for arbitrary packet header manipulations.

In order to make it more difficult for a middlebox to recalculate hashes to hide its modifications from HICCUPS, we need to include a secret into the hash that only the endpoints know. However, with no out-of-band channel between the two endpoints and no cryptographically signed key pairs with mutually agreed certificate trust lists, coming up with such a shared secret to deter overwriting of the integrity values is difficult. Instead, we look for pieces of information that would be difficult for a middlebox to know, but much easier

for one or both of the endpoints to know.

As long as one of the endpoints in a connection has such a secret, the integrity value can be encoded with that secret and a middlebox will not be able to replace it with another valid value. Although a true shared secret may not exist, we can still protect the integrity as long as the secret stays hidden from the middlebox long enough to force it to forward the initial packet. If we reveal the secret after the middlebox has already forwarded the packet for us, it will no longer be able to change the integrity and the other end host can decode the integrity value.

In AppSalt mode, HICCUPS protects integrity values by encoding them with a property of the connection that is only revealed *after* the 3WHS is complete. Such an "ephemeral secret" could be any property of a connection known only to the sender at the start of the connection. This protection technique seeks to increase the level of difficulty for *M* to make undetectable modifications.

From the perspective of the middlebox and receiving TCP endpoint, the encoded integrity values in the three HICCUPS fields remain indistinguishable from random numbers until the ephemeral secret is revealed later in the connection. Indeed, neither the receiver nor any devices along the path can determine whether or not the fields are overloaded with integrity. Thus, we are able to force a middlebox seeking to recompute our hashes to commit to a strategy *before* it even knows if the connection is HICCUPS-enabled. Since a HICCUPS-enabled TCP need not necessarily perform HICCUPS with every connection request, it is difficult for a middlebox to know when it should try to recompute new hashes. We thus add protection to the integrity while imposing as little of the increased burden as possible on the end hosts. The sending host only has to encode the integrity value and the receiving host only has to store the received integrity until the ephemeral secret is revealed.

Several different connection properties could serve as ephemeral secrets. Some possibilities include:

- A time representation of when a conversation might start
- The parties involved (IPs, ports, etc.)
- Length of a connection (time, bytes, number of packets)

- Future timing of individual packets
- Residual TTL
- Proof-of-work
- Application data

Some of these properties, such as the future timing of packets or the number of packets in a flow, are difficult to reliably control. Our HICCUPS implementation instead protects the SYN integrity values with future *application-layer content* from a data packet *yet to be sent*—an ephemeral secret that is difficult for a middlebox to reliably determine *a priori*, yet readily available for the end host wishing to initiate a new TCP connection. As in Section 5.3, the integrity values are placed in the ISN, IPID, and RCVWIN of the SYN, but now the receiving end host, as well as any middleboxes, must know the contents of future application data in order to interpret the integrity.

For the ephemeral application-layer secret, we desire to use a small portion of the data contained in the payload of the first data packet to make it simple for the receiver to locate and extract the AppSalt secret. We therefore examined the uniqueness of the initial application payload of each flow in a full day of border traffic from our organization to determine a reasonable size to use. Among application data payloads of 6,742,466 flows, we find 5,377,440 (approximately 80 percent) where the first 40 bytes are unique. The 99th percentile of the distribution is that payloads appear twice, implying that 40 bytes of ephemeral secret is a reasonable lower-bound to prevent trivial guessing. Figure 6.1 shows the distributions for various lengths across a 30 minute capture.

6.2 AppSalt Operation

To illustrate AppSalt operation, we present a common scenario where a web client connects to a web server. The client connects by performing the 3WHS and then issues an HTTP `GET` request for a specific resource. Neither the remote server nor any in-path middleboxes can reliably determine the application data at the time the SYN is observed. Only the client knows with certainty the initial HTTP application data that will be sent. In this example, the application layer data might contain such items as the `GET` uniform resource locator (URL), the host parameter, and the user agent string as shown in the example of Figure 6.2.

Since the application data needed to properly decode the SYN's integrity is not available to

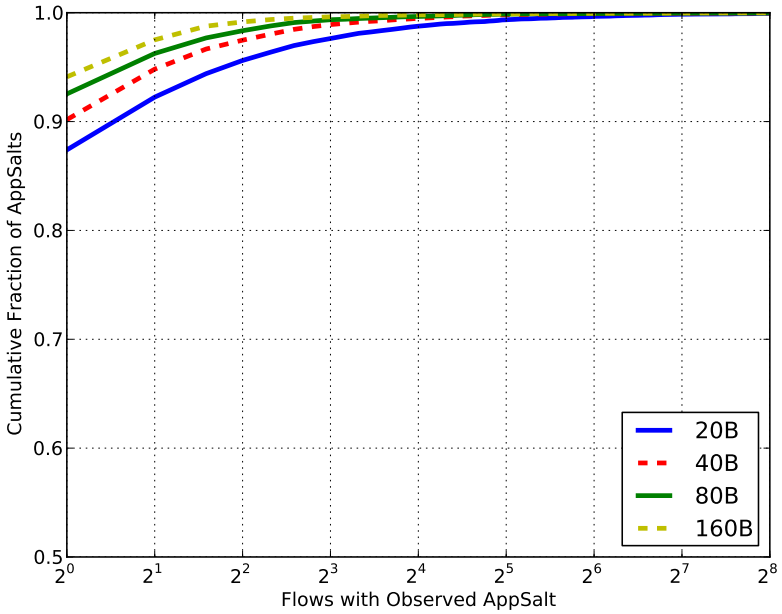

Figure 6.1: Cumulative fraction of application-layer payloads ("AppSalts") of different lengths versus number of flows in which the AppSalt appears

M at the time the SYN is received, it is difficult for *M* to make an undetectable header modification or even just to check whether the connection is HICCUPS-enabled. The ephemeral secret forces *M* to process the SYN packet before it can observe the application data. Otherwise, *M* has two remaining options if its goal is to modify the packet headers and evade detection: make a best guess of the application data, or perform a MITM attack and fake a SYN-ACK response, inducing *A* to expose the application data secret. In particular:

- *M* may attempt to guess the unseen application data (e.g., by using a profile of prior connections from *A* to *B*). However, *M* is unlikely to guess correctly for every connection between all pairs of hosts. If *M* guesses incorrectly, integrity values will not validate and the manipulations can be detected. Of course, *M* could later change the actual application data to match its guess, but doing so fundamentally alters the application-layer behavior of the connection and would most likely be readily detected since it directly impacts the user experience.
- In order to know the application data with certainty, *M* must act as a TCP-terminating proxy, a behavior that is detectable based on timing or by issuing connections to

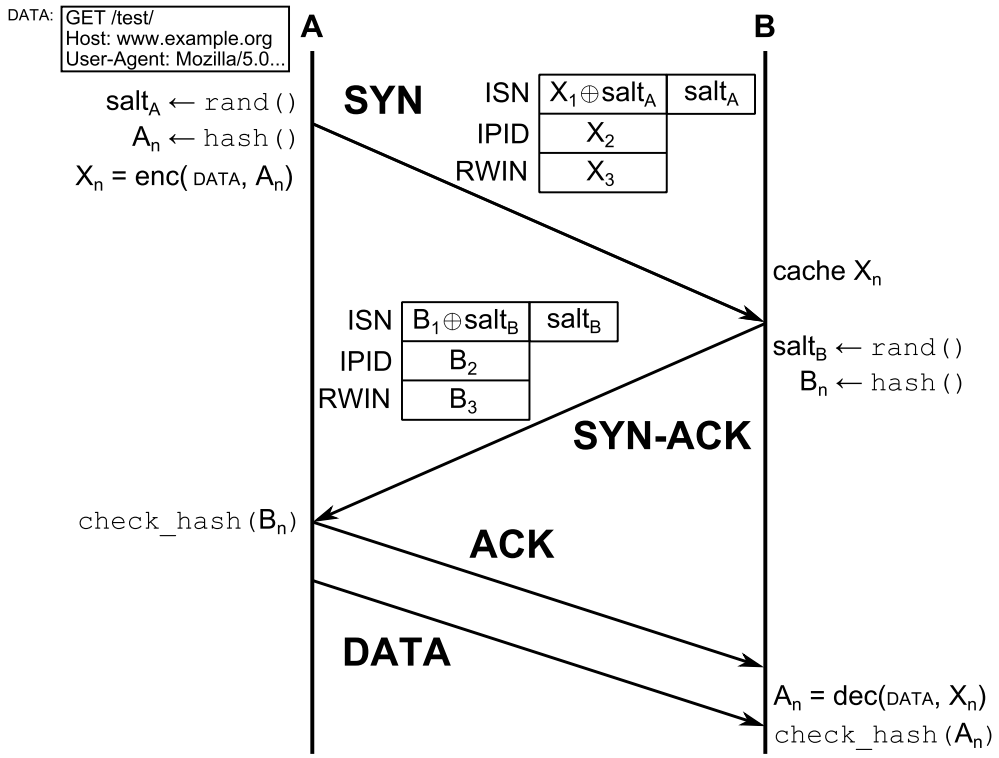

Figure 6.2: HICCUPS AppSalt protection: the integrity values in the SYN are encoded with application-layer data *yet to be sent*, forming an ephemeral secret that increases the level of difficulty for middleboxes attempting to evade HICCUPS diagnostics.

known unreachable hosts (as shown in [12]). Here, M falsely claims to be B, spoofs the SYN-ACK, and intercepts the resulting traffic. This MITM behavior permits M to rebuild the original SYN with an updated integrity value and forward it along to the true destination. The non-spoofed SYN-ACK from B must then be intercepted (assuming symmetric traffic flow, which cannot be guaranteed) and the cached data from A could be sent. This situation is more complicated than just rebuilding the integrity values; the middlebox has broken a connection and now has to marshal data between them, in addition to sending spoofed packets and buffering data. Further, the middlebox must do this for all connections, potentially representing many endpoints. More importantly, this MITM behavior to evade HICCUPS is detectable itself.

AppSalt represents our proactive approach to ensuring the continued effectiveness of HICCUPS once its algorithms and protocol become widely known. Another possible disruption technique is to perform a downgrade attack by arbitrarily overwriting all fields used by HICCUPS

for integrity. This overwriting does not circumvent the tamper-evidence, however, and the downgrade fails when there is outside *a priori* knowledge that the remote end is performing HICCUPS.

6.3 API Changes

A limitation of AppSalt is that its implementation and use requires some minor changes:

- The operating system's sockets API must allow for an application to provide data before initiating the TCP 3WHS.
- Depending on how the operating system's socket calls were modified, any applications that wish to use AppSalt must adjust their TCP connection system calls.

Traditionally, in most operating systems a client TCP issues a series of socket calls: `socket()`, `connect()`, and then `send()`. However, with AppSalt, `connect()` cannot be called first as it will initiate the 3WHS and send the SYN before the kernel has the necessary application data over which to calculate integrity.

In Chapter 7, we discuss the details of how we implemented AppSalt in the Linux kernel. Our AppSalt implementation requires that applications specifically request its protections by passing a special flag to the operating system in its socket calls. Our TCP still enables standard HICCUPS for all TCP connections, however, by default and without any application changes.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Implementation and Validation

If you hold a cat by the tail you learn
things you cannot learn any other way.

Mark Twain

We have implemented HICCUPS along with our integrity protection scheme, AppSalt, as a patch for Linux kernel version 3.9.4 [112]. We selected this version for implementation simply because it was the latest stable version at the time we began our work, but any necessary changes to be able to apply the patch to a newer version should be minimal. We have archived a copy of our patch as it was when we conducted the experiments in this dissertation [113]. The most recent version can be found online [114].

7.1 Implementation overview

Our implementation of HICCUPS modifies the existing TCP/IP stack of the Linux kernel to augment outgoing packets with integrity and perform special processing of incoming packets to perform the necessary integrity checks. Once a kernel has been patched with HICCUPS, it can perform integrity checks with other HICCUPS-capable hosts around the Internet and process the results within TCP. Our desired end goal is for HICCUPS to one day be brought into the mainline Linux kernel so no patching would be needed in order to make a system HICCUPS-enabled.

In order to give an idea of the size of our kernel patch implementation, we cataloged the lines of code used and note that the core integrity transmission and protection protocols require just over 700 lines, including changes to both source and header files. Table 7.1 shows the complete breakdown of the lines in our patch by their associated component functionality.

Core HICCUPS refers to the integrity transmission and validation components. AppSalt is the protection mechanism. In our implementation, we chose to leverage both the CRC32

	Blank lines	Comments	Code
Core HICCUPS	111	245	560
Debugging	52	72	299
Faking options	4	6	43
AppSalt	33	41	164
Murmur3	51	92	151
Total	251	456	1217

Table 7.1: Lines of code broken down by component functionality

and Murmur3 [111] hashes within HICCUPS. Since Murmur3 was not already in Linux, we added it, which took about 150 lines of code. The options faking code is only used by our evaluation measurements to imitate the Multipath TCP `MPCAPABLE` [21] response in SYN-ACK packets.

7.2 HICCUPS Linux API

User-space applications can request that a specific HICCUPS coverage type be used for their connection by issuing a `setsockopt()` call prior to opening the connection. Listing 7.1 shows an example connection attempt where an application specifies its desired HICCUPS coverage type (see Section 5.5 for a list of coverages to which `cvr` can be set).

Listing 7.1: Requesting a coverage type

```
sockfd = socket(...);
setsockopt(sockfd, SOL_TCP, TCP_HICCUPS_COVERAGE, &cvr,
    sizeof (cvr));
connect(sockfd, ...);
write(sockfd, msg, msglen);
close(sockfd);
```

Similarly, once a connection has been established an application can read the results of the HICCUPS integrity checks by issuing some calls to `getsockopt()`. A total of three calls to `getsockopt()` are required in order to retrieve all status values: the overall HICCUPS match status, the SYN match bits, and the SYN-ACK match bits. Listing 7.2 shows some example connection code.

Listing 7.2: Retrieving the HICCUPS status of a connection

```
sockfd = socket(...);
connect(sockfd, ...);
write(sockfd, msg, msglen);
getsockopt(sockfd, SOL_TCP, TCP_HICCUPS_STATUS, &result1,
    &result1_len);
getsockopt(sockfd, SOL_TCP, TCP_HICCUPS_SYN_MATCH, &result2,
    &result2_len);
getsockopt(sockfd, SOL_TCP, TCP_HICCUPS_SYNACK_MATCH, &result3,
    &result3_len);
close(sockfd);
```

AppSalt

As we detailed in Chapter 6, AppSalt is an optional layer of protection that requires the kernel to know the initial byte range of data that an application wishes to send in a connection before the TCP 3WHS is initiated. This situation is incompatible with the traditional ordering of Linux socket calls as shown in Listing 7.3. In that sequence of calls, the TCP 3WHS is initiated by the `connect()` call, and the SYN is sent before the kernel is presented with any application data.

Fortunately, this problem has already been approached in Linux and a good parallel exists for requiring data at the time of connection initiation, TCP Fast Open (TFO). TFO [23] is a TCP enhancement that allows follow-up connections between two endpoints to not have to wait for the full 3WHS to complete. If two TFO-enabled TCPs have communicated previously, they shared a TFO cookie (a cryptographic token sent in a new TCP option kind) that can be sent with subsequent connections to authorize immediate data transmission and save the latency cost of the additional RTT spent waiting for the SYN-ACK to be received. Programs that use TFO initiate all connections using `sendto()` or `sendmsg()` with the `MSG_FASTOPEN` flag, as opposed to the typical `connect()` and `send()` sequence. In this way, the kernel can embed data in the SYN for connections with a valid TFO cookie.

We therefore leverage the same socket API changes as TFO to allow a client program to request AppSalt-mode HICCUPS. Primarily, we add a new message flag within the

framework established by TFO that can be used by the `sendto()` call: "`MSG_HICCUPS`." Listing 7.4 shows the new sequence of calls that applications use in order to request AppSalt protection. Note that now there is no `connect()` call required for socket setup. As soon as an application wants to send data it issues the `sendto()` and all of the AppSalt logic along with TCP connection setup happens behind the scenes in the kernel.

Listing 7.3: Old socket call order

```
s = socket(...);
connect(s, addr);
send(s, msg);
```

Listing 7.4: New socket call order

```
s = socket(...);
sendto(s, msg,
  MSG_HICCUPS, addr);
```

As an added benefit of the TFO-style approach, the addition of HICCUPS support is trivial for applications that already support TFO (e.g., Google Chrome [115]). Since we use the same calling method as TFO with just a different flag name, it is trivial for applications that already use TFO to also use AppSalt protection with HICCUPS. The application need only `OR` the `MSG_HICCUPS` flag with the `MSG_FASTOPEN` flag in its `sendto()` calls, as shown in Listing 7.5. Any time that application data cannot be used (i.e., a program does not use the new socket API or it is a TFO connection with data in the SYN) the ISN contains an integrity value without the application-layer obfuscation (as in Figure 5.1).

Listing 7.5: TCP Fast Open with HICCUPS

```
s = socket(...);
sendto(s, msg, MSG_FASTOPEN | MSG_HICCUPS, addr);
```

An alternate strategy that could be used to implement AppSalt would be to instead modify the kernel API logic behind the `connect()` call so that the 3WHS is not initiated until the first `send()` call. This change would have the positive property of automatically engaging AppSalt protection for all applications without having to update them, but concerns about compatibility and decreased acceptance by the community led us to opt for the more gradual approach. We also desire the approach that gives the application designer more explicit control in case there is some unforeseen problem with delaying connection initiation.

Given the firmly established history and wide understanding of socket calls, we believe the best method of implementing AppSalt is the TFO-style approach.

Finally, we feel it is important to reiterate that normal HICCUPS as presented in Chapter 5 works fine for all connections in Linux without any of the previously described API changes. All of these changes are to enable the AppSalt enhanced protection mode as described in Chapter 6. When an application does not support the `MSG_HICCUPS` flag, we emphasize that nothing breaks, the only result is that that application's connections use the regular HICCUPS without the additional ephemeral secret protection.

7.3 HICCUPS Details

In order to implement HICCUPS in the kernel, we add hooks at key places where SYN and SYN-ACK packets are processed on both the incoming and outgoing paths. Table 7.2 lists our HICCUPS functions and hook placements corresponding to each type of SYN or SYN-ACK related event. Locations for the hooks were chosen through a combination of code analysis, debugging, and trial and error. The hook locations describe files under the `net/ipv4` subdirectory of the Linux kernel source tree, and the line numbers represent the locations within each file prior to editing [112, 113].

Event	HICCUPS function	Hook location
SYN sent	`tcp_hiccups_syn_out`	`ip_output.c:403`
	`tcp_hiccups_after_syn_out`	`tcp_output.c:3029`
SYN received	`tcp_hiccups_syn_in`	`tcp_ipv4.c:1515`
SYN-ACK sent	`tcp_hiccups_synack_out`	`ip_output.c:161`
	`tcp_hiccups_after_synack_out`	`tcp_ipv4.c:863,1652`
SYN-ACK received	`tcp_hiccups_synack_in`	`tcp_input.c:5724`

Table 7.2: HICCUPS functions and hooks, by TCP event

A major challenge working within the Linux kernel is that sequence numbers and IPID values are selected before much of the final packet is built. In particular, since the TCP initial sequence number does not originally depend on the full packet or any information at the IP layer, it can be calculated early on. However, with HICCUPS, we change this design and make the sequence number a function of other fields in the packet header. Therefore, we must postpone calculation of the ISN until the full packet has been created by the kernel and we know the values of all fields we wish to cover. Once the SYN packet has been sent,

we go back into the socket structures and change the originally stored initial sequence number to a HICCUPS-enabled sequence number. This changeback need only occur once in the connection, immediately after sending the SYN packet and returning up through the layers to TCP.

The two "after" hooks listed in Table 7.2 are also due to the challenge of the kernel needing and using the sequence number before we have the full packet available for hashing. At the point in the code flow at which the full packet is available, we are often working with clones of the socket buffer and need to wait until we come back up the transport layer to update the rest of the sequence number fields.

7.3.1 Server-side overhead

We quantified the server-side overhead associated with HICCUPS using the Linux kernel's ftrace facility [116]. ftrace contains a function graph tracer that is able to observe entrance and exit events of functions. To track the exiting of functions, the tracer overwrites the assembly-level return address of each function with its own custom return code. After it processes the function return event, it returns execution to the original return address of the function.

Table 7.3 shows the results from tracing the full function graph with ftrace associated with an incoming SYN/ACK. The table shows the full ordering of kernel function calls leading up to the primary call to `tcp_v4_conn_request`. Table 7.4 shows the ordering of calls within `tcp_v4_conn_request`, along with their associated run times for our HICCUPS-enabled system. Important to note in the table is the fact that our primary function call, `tcp_hiccups_syn_in`, is similar in run time to other sub-functions of the connection request call. For example, `security_inet_conn_request` handles SELinux processing for the incoming SYN and takes about 30 percent longer to run.

In order to get a true value for the total overhead involved in running HICCUPS, we compare the run times of the lowest link layer function shown in Table 7.3, `net_rx_action`. We must take this approach because, as noted in the caption of Table 7.4, ftrace results can be inaccurate for functions that contain many sub-functions that give execution back to the function tracer. By configuring ftrace to filter only for function calls to `net_rx_action`, we should obtain the most accurate run time measurements as possible.

Layer	Function name
TCP layer	tcp_v4_conn_request
	tcp_rcv_state_process
	tcp_v4_do_rcv
	tcp_v4_rcv
IP layer	ip_local_deliver_finish
	ip_local_deliver
	ip_rcv_finish
	ip_rcv
Link layer	__netif_receive_skb_core
	__netif_receive_skb
	netif_receive_skb
	napi_gro_receive
	e1000_clean_rx_irq
	e1000_clean
	net_rx_action
Non-network specific	__do_softirq
	irq_exit
	do_IRQ
	common_interrupt
	default_idle
	cpu_idle
	rest_init
	start_kernel
	i386_start_kernel

Table 7.3: The function call graph generated by ftrace upon receipt of a SYN. The graph is shown up to tcp_v4_conn_request, which handles processing of a new connection.

Taking the average over 1,000 connection attempts, we compared the total time spent processing a SYN/ACK between the HICCUPS-patched kernel and an unmodified, or "vanilla", kernel. Table 7.5 shows the statistics we computed across the 1,000 connections. We found that the average overhead added by our unoptimized implementation is about 8.5 percent of the compute time in the vanilla kernel.

7.3.2 SYN cookies

As discussed in Section 4.7.6, the Linux kernel implements SYN cookies but does not use them under normal non-attack conditions. For all connections, Linux only resorts to SYN cookies when it believes it is under a SYN flooding attack [106].

Latency	Function name
	`tcp_v4_conn_request() {`
2.214 us	`kmem_cache_alloc();`
5.864 us	`tcp_hiccups_syn_in();`
2.284 us	`tcp_parse_options();`
7.579 us	`security_inet_conn_request();`
4.353 us	`inet_csk_route_req();`
4.000 us	`tcp_make_synack();`
1.777 us	`__tcp_v4_send_check();`
+ 41.680 us	`ip_build_and_send_pkt();`
1.815 us	`tcp_hiccups_after_synack_out();`
2.640 us	`inet_csk_reqsk_queue_hash_add();`
+ 133.828 us	`}`

Table 7.4: The durations of sub-functions within the call to tcp_v4_conn_request, which handles processing of a new connection. The + sign indicates that the latency is inaccurate due to additional overhead from repeated calls to the ftrace probing code.

HICCUPS Linux w/ AppSalt enabled		Vanilla Linux	
min	84.446 μs	min	84.075 μs
max	635.666 μs	max	490.836 μs
median	119.634 μs	median	110.533 μs
mean	124.037 μs	mean	114.254 μs
var	903.493 μs	var	697.618 μs
std	30.058 μs	std	26.412 μs

Table 7.5: Comparison of run times for the net_rx_action function call between a HICCUPS-enabled kernel with AppSalt enabled and an unmodified kernel. The table shows statistics across 1000 incoming connections.

With our implementation of HICCUPS, we maintain this behavior and allow the kernel to switch over to SYN cookies as system resources become strained due to a possible SYN flood attack. The failover method is very appealing since, under normal conditions, a system would rather not use SYN cookies due to the limits on what options it can support, and under attack conditions, a TCP would likely rather not be doing HICCUPS with its small but additional hash overheads. Continuing to use the same failover method for SYN cookies is most advantageous to the server.

7.4 Testing in a Controlled Environment

Before beginning the measurements described in Chapter 8 using HICCUPS, we first performed a series of experiments to empirically vet our implementation and ensure that both

the methodology and coding were correct. We deployed HICCUPS-enabled hosts to a virtualized testing environment for initial validation purposes.

Using virtual hosts running inside Virtualbox, we model a situation where two end hosts are connected with a third system along the path between them. The layout is shown in Figure 7.1. Each of the two end hosts are running the HICCUPS Linux kernel and the third system acts as a transparent middlebox in their communications.

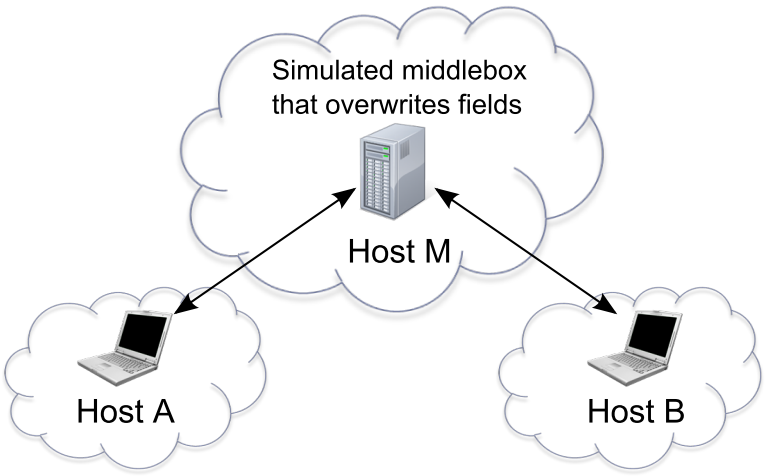

Figure 7.1: Diagram of Virtualbox testing

In order to imitate some of the possible modifications that middleboxes make, we use an `iptables` rule to redirect each forwarded packet up to user-space via the nfqueue-bindings software [117]. A Scapy [118] script written in Python receives the packet, modifies it, recalculates the network checksum, and then forwards the packet on to its destination. We have written the Scapy script so that it can make an array of common modifications to packets as shown in Table 7.6.

Using the middlebox script, we tested the effectiveness of the HICCUPS Linux kernel to detect each of the packet header modifications introduced by our synthetic middlebox. Virtual machines running the HICCUPS kernel performed 50,000 trials that established 3.2 million total TCP connections, with each of these connections traversing the middlebox simulator. A total of four different modification configurations were tested within the 3.2 million connection probes. Automated verification found that HICCUPS properly inferred

Field modified	Description of modification
IP ECN codepoint	If an ECN-capable codepoint is set, zero it out.
	If a congestion-experienced codepoint is set, set one of the bits to zero.
IP DF flag	Complement the Don't Fragment bit
TCP ECN flags	Set both flag bits to zero
TCP ECN flags	Complement each ECN flag bit
IP ID	Set value to zero
TCP ISN	Translate values during the handshake
TCP MSS	If no MSS is set, set a random one between 1 and 1460 bytes
	If an MSS is set, lower it to a specified value
TCP Window Scale	If no window scale is set, set one at a factor of 7
	If a window scale is set, change it to a factor of 7
Reserved fields	Turn reserved bits in TCP and IP headers to 1
TCP Receive Window	Offset the receive window by a specified value
All	All of the above modifications enabled at once

Table 7.6: List of modifications made by middlebox simulator

the path behavior for 100 percent of the connections, meaning that a change was properly detected when it was made, and the hosts did not detect any changes when no header modifications were introduced by the simulator.

7.5 Demonstration of Debugging with HICCUPS

For the purposes of showing how HICCUPS can be used in debugging, we will walk through an example scenario to detect a blocked ECN negotiation. In this scenario, both hosts *A* and *B* of Figure 7.1 are ECN-enabled and request ECN during TCP connection negotiation. The middlebox script described in Section 7.4 is programmed to set both ECN flags in the TCP header to zero on any packets it sees (i.e., to improperly clear these ECN bits). As we will see later in Section 8.3.2, this behavior occurs in the wild on the actual Internet. The modification has the effect of preventing both end hosts from using ECN, even when they both support it.

Working from the point of view of Host *A*, we have reason to believe that ECN is not working properly. For example, if we run a packet capture during a connection attempt to Host *B*, we notice that SYN-ACKs returning from Host *B* do not have ECN enabled, as shown in Figure 7.2. The SYN leaves Host *A* with the ECE and CWR TCP flags set

(to indicate a desire to enable ECN for this connection), but the SYN-ACK returns from *B* with neither flag set, even though they should be set. If we have administrative access to Host *B*, we can verify this claim using traditional debugging methods (i.e., run a packet capture at both ends and compare the headers). However, with HICCUPS, we can verify this behavior without needing access to Host *B* at all.

```
demo@Host_A: ~ $> sudo tcpdump -i p7p1 "tcp[tcpflags] & tcp-syn != 0"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on p7p1, link-type EN10MB (Ethernet), capture size 65535 bytes

17:04:51.216901 IP hosta.operator.net.rrac > hostb.remote.net.http: Flags [SEW],
 seq 148508632, win 1094, options [mss 1460], length 0
17:04:51.239197 IP hostb.remote.net.http > hosta.operator.net.rrac: Flags [S.],
seq 980663579, ack 148508633, win 36086, options [mss 1460], length 0
```

Figure 7.2: Screenshot of packet capture from Host *A*

Using HICCUPS, we confirm that a middlebox is disabling our ECN flags in transit between *A* and *B*. Shown in Figure 7.3 is the output from running our user-space HICCUPS test client on Host *A*. The user-space HICCUPS client mirrors the same logic used in the Linux kernel patch implementation.

```
demo@Host_A: ~/hiccups/client_rawsock/src $> sudo ./hc -e hostb.remote.net
-------------------------------------------------
| TCP HICCUPS Raw Socket Client - version 2.3     |
| Written by: CMAND, hiccups@cmand.org            |
-------------------------------------------------

HICCUPS ability check: PASS -- HICCUPS works well on this path!

Modifications Detected:
MOD: ECN TCP modified on SYN

** Done probing **
Number of probes used: 9
Reporting successful.  Thank you for your contribution!
demo@Host_A: ~/hiccups/client_rawsock/src $>
```

Figure 7.3: Screenshot of HICCUPS test client probing from Host *A* to *B* with ECN enabled

First, HICCUPS probes (i.e., standard TCP connection attempts) are sent from Host *A* to *B*. The failed integrity checks reveal to *A* that a modification is occurring on its SYN, but not the SYN-ACK. Specifically, that the modification is to one of the bits covered by

the `HECNTCP` probe type. Second, assuming we have no knowledge of what changes the middlebox simulator is making, we can try disabling ECN to further test if a middlebox is targeting ECN in particular. We disable ECN negotiation on Host *A* and try testing with the HICCUPS client again. Now, as we see in Figure 7.4, no modifications were detected by HICCUPS implying that the middlebox was specifically configured to target ECN flags.

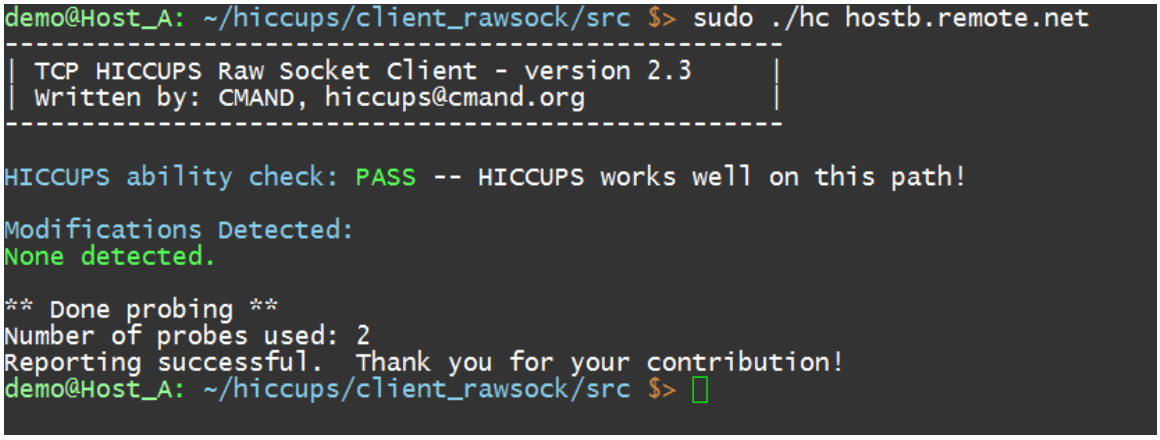

Figure 7.4: Screenshot of HICCUPS test client probing from Host *A* to *B* without ECN enabled

CHAPTER 8:
Surveying Internet Paths with HICCUPS

If you cannot measure it, you cannot improve it.

Lord Kelvin

This chapter details results from running HICCUPS in the wild. We examine the types, frequencies, and symmetry of HICCUPS-inferred modifications and give examples of how a HICCUPS-enabled TCP can adjust its behavior based on path inference to improve performance. Last, we discuss HICCUPS overhead, including the empirical number of RTTs for full-path characterization.

While previous research [12, 19, 20, 39, 43] examined real Internet paths to catalog various forms of packet header modifications, these efforts required some degree of interaction external to the operating systems. To our knowledge, HICCUPS is the first solution to both capture measurements of packet header modifications within TCP and expose the results *directly through the operating system itself.* For example, the servers in our measurement infrastructure do not run any specialized server application. Instead, we simply start a standard HTTP daemon that listens on the desired port(s). With a HICCUPS-enabled kernel, no extra support is required to perform HICCUPS and expose path behaviors to the operating system and applications.

8.1 Experimental Infrastructure

Using HICCUPS-enabled hosts, we survey a diverse set of real Internet paths. We employ 218 Planetlab [119] nodes, 56 Archipelago (Ark) [120] nodes, and 12 distributed HICCUPS servers; the autonomous system (AS) and geographic distribution of our infrastructure is given in Tables 8.1 and 8.2. This infrastructure enables us to run HICCUPS between 3,288 pairs of distinct hosts, testing 26,304 directed path/port pairs.

Our HICCUPS-enabled Linux kernel runs on 12 systems: one at MIT, one at Virgina Tech, one at ICSI, one on Comcast Business, and one at each of the eight Amazon EC2 infras-

Table 8.1: Top ASNs represented

Servers		PlanetLab		Ark	
AS16509	6	AS680	13	AS22773	3
...	1 ea.	AS2200	6	AS1213	2
		AS766	6	...	1 ea.
		...	<6 ea.		
Total	7	Total	154	Total	53

Table 8.2: Geographic distribution

Location	PlanetLab	Ark	Servers
Europe	101	18	1
N. America	75	25	7
Asia	26	9	2
S. America	10	1	1
Oceania	6	0	1
Africa	0	3	0
Total	218	56	12

tructure sites. To run HICCUPS from PlanetLab (where installing a custom Linux kernel is not possible), we duplicate the connection initiation portion of TCP with HICCUPS into a user-space client that employs raw sockets to craft HICCUPS-enabled SYNs.

In selecting PlanetLab nodes, we used PlanetLab's management API to use a single node per site. Thus, all 218 Planetlab nodes we use represent distinct sites. The PlanetLab nodes were distributed both geographically and logically around the Internet. The Planetlab and Ark nodes reside in 207 distinct ASes. Geographically, our Planetlab nodes are situated in five continents and 37 different countries, while the Ark nodes are spread across 28 countries.

8.2 Experimental Parameters

From each PlanetLab and Ark vantage point, we execute SYN exchanges with each server on four different TCP ports to capture port-specific behavior: 22, 80, 443, and 34343. The first three are common service ports; port 34343 is used for consistency with Honda *et al.* [19]. We send 16 SYNs to each of the four ports, with each SYN covering one of the different coverage types listed in Table 5.4. Note that not all paths require all 16 connections to fully ascertain the path conditions from HICCUPS; about 90 percent of

paths can be fully characterized in two RTTs. We examine this aspect further in Section 8.4.

To make middlebox modification behaviors visible, we must enable different TCP and IP extensions (such as those discussed in Section 2.1.3) during the connection setup. Table 8.3 lists the sets of options we use in our experiments, including MSS, SACK permitted, Window Scale, Timestamp [52], Multipath TCP MPCAPABLE [21], and a non-standard experimental option with a kind value of `99`.

Table 8.3: Experimental parameters for each trial

Trial	MSS	ECN	SACK Permit	Win Scale	Time stamp	MP-TCP	Exp
1	1460		Y	7	Y		Y
2	1460		Y	7	Y	Y	
3	1460		Y	7	Y		
4	1460	Y					
5	480						
6	1460						
7	1600						
8	None						

8.3 Detected Modifications

Following the inference procedure in Section 5.3 and Tables 5.1 and 5.2, we use HICCUPS to detect a variety of packet header manipulations. If a probe passes integrity checks at the receiver and the forward path status bits return intact, the TCP initiator infers that its packets (on the forward path) arrive without modification. Similarly, if the integrity checks on the SYN-ACK match, the initiator infers that the reverse path does not modify headers. All data we present in this chapter comes from the clients on PlanetLab and Ark.

Figure 8.1 displays the cumulative fraction of probes per host with passing integrity versus the fraction of nodes (PlanetLab and Ark nodes combined with NAT results excluded). The common case is that both the forward and reverse paths experience no modifications. For approximately half of the nodes, all probes match integrity, while approximately 80 percent of the nodes have 99 percent or their probes match integrity. The distributions for the two asymmetrical integrity results are visible in the lower-right of the figure. Approximately 80 percent of nodes never experience this case.

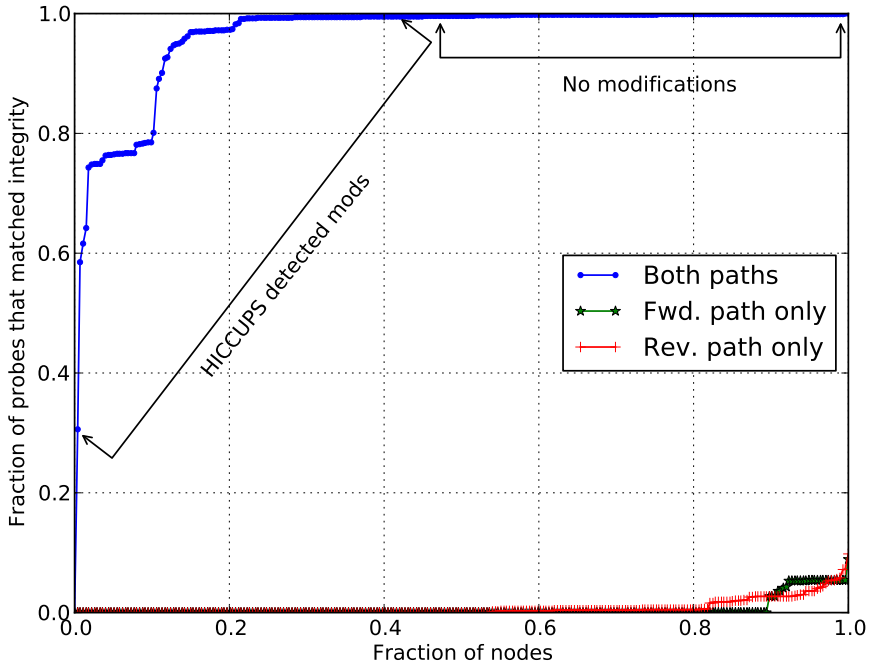

Figure 8.1: Distribution of matching probes, by direction. NAT modifications have been excluded.

Table 8.4 summarizes the probe results according to coverage type. The most common modification is paths that add or change MSS values. The `HNAT`—and consequently the inclusive `HFULL` probe—fails for the large majority of paths. This is unsurprising as address translation is performed near the server for nine of our 12 servers. We verified that while our Amazon EC2 servers experienced NAT, they made no other header modifications. Tables Table 8.5 and Table 8.6 show the results of running our detection logic on the path, with each modification broken out by line. A modification is marked as an addition if the change occurred when we did not request the option in the probe. In the following subsections, we more closely examine specific modifications.

8.3.1 ISN translation

We find incidences of sequence number translation in tests from 24 of 218 Planetlab nodes (11.0 percent). The ISNs are translated in both directions on 20 nodes, while for four nodes, just the forward path translates sequence numbers. Only one of the Ark nodes is subject to ISN translation that occurs on forward path only.

Table 8.4: Summary of results by coverage type

	Integrity Match				
Coverage	Both	Fwd	Rev	Neither	Timeout
HFULL	21867	597	985	80931	836
HNAT	25286	2	0	79129	799
HNONAT	91214	2397	2459	8329	817
HNOOPT	100535	71	2050	1732	828
HONLYOPT	92948	2542	1162	7736	828
HECNIP	102066	69	1693	572	816
HECNTCP	103777	10	47	585	797
HLEN	103451	17	359	574	815
HMSS	93365	2545	855	7632	819
HWINSCL	103685	16	5	690	820
HTSTAMP	103834	27	7	539	809
HMPTCP	103023	20	837	551	785
HEXOPT	102907	12	888	564	845
HFLAGS	102591	18	76	1719	812
HSAFE	103824	16	0	551	825
HNULL	103752	21	0	563	880
Total	1458125	8380	11423	192397	13131

The frequent occurrence of sequence number translation motivates in part our choice to use three hash fragments, as detailed in Section 5.3. If, for instance, the ISN alone carried integrity, HICCUPS would not work for 25 of our 274 nodes and we would be unable to detect any header modifications beyond ISN translation. In contrast, HICCUPS can withstand a single modification to any one of the three integrity-carrying fields (i.e., ISN, IPID, and receive window).

However, should any pair of the three fields be modified, HICCUPS loses the capability to detect specific field modifications, only noting that a change occurred to at least one pair of the three integrity fields. Tables 8.5 and 8.6 list paths where this behavior occurs under the heading "HICCUPS not capable." 68 flows from PlanetLab (0.7 percent) and four flows from Ark (0.2 percent) saw two or more integrity fields overwritten. Since we control all of the nodes, we performed post-mortem analysis of packet captures taken during the measurements and see that the TCP receive window is artificially lowered in-path. In practical use, however, HICCUPS cannot obtain any fine-grained information for

Table 8.5: Summary of HICCUPS-inferred header modifications on PlanetLab. Detection of ISN, IPID, and receive window modifications are mutually exclusive to HICCUPS. If two or three occurred, it registered as "HICCUPS not capable" instead.

Change	Both	Fwd	Rev	Flows	Affected
HICCUPS not capable	68	0	2	10360	0.68%
NAT	7704	0	0	10281	74.93%
ISN translation	924	178	0	10290	10.71%
IPID change	0	0	0	10290	0.00%
RCVWIN change	0	0	0	10290	0.00%
ECN IP add	26	0	0	10270	0.25%
ECN IP change	16	1342	48	10283	13.67%
ECN TCP add	16	0	0	10261	0.16%
ECN TCP change	19	46	0	10285	0.63%
MSS add	119	47	1036	10258	11.72%
MSS480 change	21	0	1132	10281	11.21%
MSS1460 change	1113	0	0	10275	10.83%
MSS1600 change	1105	157	0	10294	12.26%
SACK Permit changed	1	24	0	10123	0.25%
Timestamps add	12	0	0	10267	0.12%
Timestamps change	26	2	0	10279	0.27%
Window Scaling add	45	0	0	10265	0.44%
Window Scaling change	24	0	0	10279	0.23%
`MPCAPABLE` change	24	837	0	10267	8.39%
Exp. option change	20	884	0	10266	8.81%

such paths.

8.3.2 ECN

We monitor behavior of the ECN fields in both the IP and TCP headers. Figure 8.2 shows the results of each probe arranged by host in the combined PlanetLab and Ark datasets. Each of the three plots in the figure represents the results from probing each of the 48 server ports from each of the 274 nodes. Each plot is sorted so that primary result types are grouped together. The first plot shows the behavior when ECN was disabled, while the lower two show behavior after ECN has been enabled. While ECE and CWR TCP flags are rarely affected (we only saw such mods on paths from one PlanetLab node), modifications to the IP codepoint are more common. We observed about 13 percent of paths on both PlanetLab and Ark would zero the codepoint if it were enabled. None of the Ark nodes

Table 8.6: Summary of HICCUPS-inferred header modifications on Ark. Detection of ISN, IPID, and receive window modifications are mutually exclusive to HICCUPS. If two or three occurred, it registered as "HICCUPS not capable" instead.

Change	Both	Fwd	Rev	Flows	Affected
HICCUPS not capable	4	0	0	2684	0.15%
NAT	2114	0	0	2677	78.97%
ISN translation	0	48	0	2680	1.79%
IPID change	0	0	0	2680	0.00%
RCVWIN change	0	0	0	2680	0.00%
ECN IP add	2	0	0	2664	0.08%
ECN IP change	11	342	0	2675	13.20%
ECN TCP add	6	0	0	2670	0.22%
ECN TCP change	16	0	0	2675	0.60%
MSS add	10	96	140	2668	9.22%
MSS480 change	5	0	139	2674	5.39%
MSS1460 change	134	12	12	2678	5.90%
MSS1600 change	140	154	12	2672	11.45%
SACK Permit changed	0	0	0	2667	0.00%
Timestamps add	9	0	0	2669	0.34%
Timestamps change	10	0	0	2672	0.37%
Window Scaling add	9	0	0	2665	0.34%
Window Scaling change	5	0	0	2669	0.19%
`MPCAPABLE` change	8	0	0	2673	0.30%
Exp. option change	13	0	0	2676	0.49%

were found to exhibit changes.

8.3.3 Application Performance

An important consequence of HICCUPS is that knowledge of the end-to-end header modification state of a path can improve the performance of applications that depend on TCP.

For instance, in the case of sequence number translation that is SACK-naïve, performance suffers in proportion to loss rate [53]. For ECN, performance suffers when false congestion signals are inadvertently marked, experiencing dramatic performance impact if a congestion codepoint is added, or a TCP-layer congestion echo is added [20]. To highlight the potential impact on TCP performance, we examine a particular effect, observed in the wild, in detail.

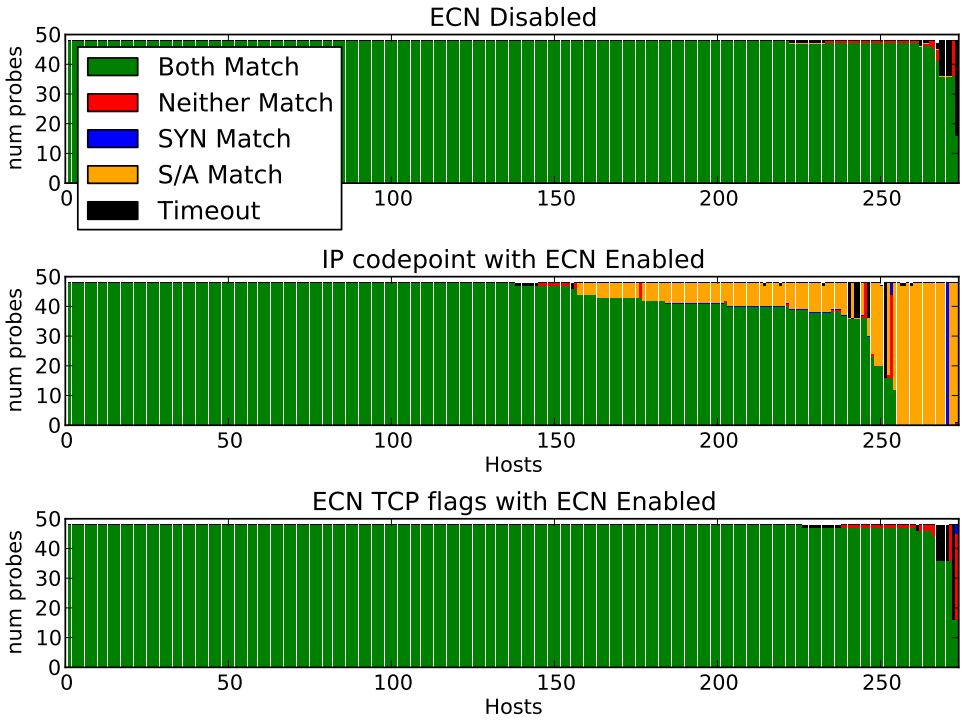

Figure 8.2: Distribution of HICCUPS-inferred ECN path properties. For the IP codepoint, HICCUPS only notes a change to the OR of the bits (Section 5.5).

We find a PlanetLab node (planetlab2.mta.ac.il) where the forward communication transparently adds a TCP window scale value of seven to the SYN, but the reverse path strips the window scale by replacing it with four NOP options in the returned SYN-ACK. The behavior is destination port-specific: it did not occur on connection attempts to ports 22 or 34343, only to 80 and 443. Ultimately, one end of the communication believes that window scaling negotiation has occurred, while the other does not.

We perform bulk transfer to the node performing window scaling and observe that the traffic is flow controlled—the receiver is sending scaled values in the receive window, but the sender interprets those values as unscaled. HICCUPS informs us of the option mangling and we disable window scaling. Our performance tests reveal a dramatic difference where the throughput more than doubles without window scaling since the congestion window can open more than one or two MSS. We alerted the operator of the node and they were unaware of the behavior. Further investigation revealed the issue was with a system in their

provider's network.

8.4 Expected Interactions Required

Across real paths in our PlanetLab and Ark datasets, we calculated the number of TCP interactions it would take for two HICCUPS hosts to fully ascertain the path header modification state. For PlanetLab, our dataset contained 83,712 flows with 261,185 total SYN exchanges were required to fully explore the space of header modifications with HICCUPS. This amounts to an average of 3.1 SYN exchanges per flow. For Ark, we required 58,083 SYN exchanges across a total of 21,504 flows, for an average of 2.7 exchanges per flow.

Figure 8.3 shows that about 85 percent of flows were able to fully determine the modifications of their paths after checking just `HNONAT` and `HFULL`. Should NAT detection not be desired, the check for `HFULL` could be omitted from the strategy shown in Section 5.6, further reducing the required number of probes.

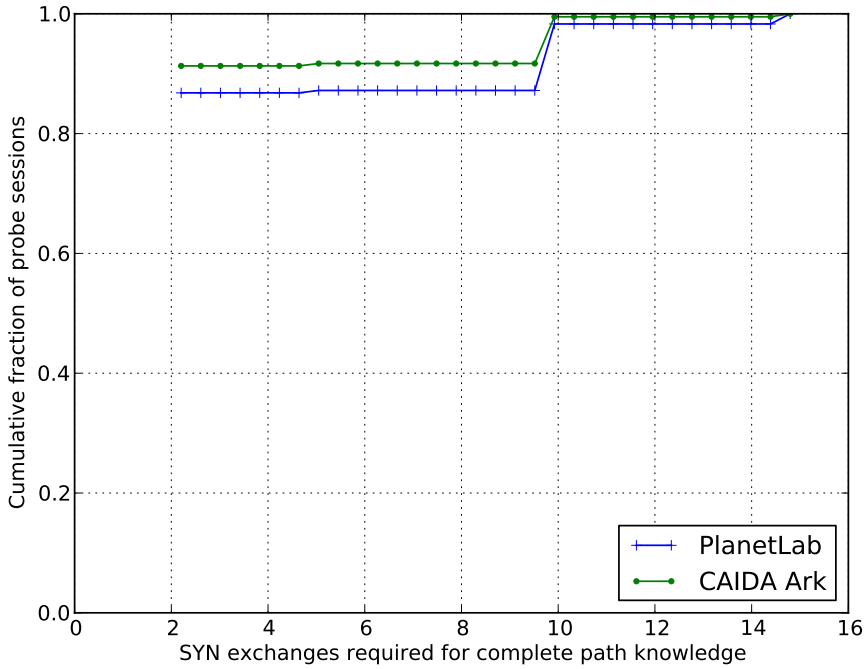

Figure 8.3: Empirical HICCUPS RTTs required for complete path properties inference

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 9:
Conclusions and Future Work

The best time to plant a tree was 20 years ago.
The second best time is now.

Chinese Proverb

Detecting issues with protocol interactions that impact network performance is a difficult process, complicated by a heterogeneous collection of standalone, opaque, and difficult-to-configure middleboxes. As we have shown in previous chapters, such issues are often subtle and—until now—required a manual debugging process by a trained administrator in order to identify and correct problematic middleboxes. In the meantime, TCP performance can needlessly suffer when using an extension that is disrupted along a path.

In this dissertation, we present TCP HICCUPS, a backward-compatible and incrementally deployable extension to TCP that automates the question of how a TCP is being interpreted at the remote endpoint. To do so, HICCUPS applies a tamper-evident seal to the packet headers of the TCP 3WHS, revealing header manipulation to both sides of a TCP.

The information provided by HICCUPS allows TCP endpoints to infer when specific options and extensions are unsafe for use along a path, and gracefully fallback to legacy TCP. In other words, a TCP can use the information from HICCUPS to selectively toggle protocol extensions that best fit a given path, maximizing application performance in the presence of misconfigured, non-standard, or legacy network middleboxes. In this fashion, we take a step toward enhancing the cooperation between end hosts and middleboxes that may exist along a path across different administrative boundaries. For example, we show how HICCUPS actively helped detect a subtle, performance-impacting condition with window scaling on a real Internet path of which the operator was unaware. HICCUPS helped the system in this case achieve twice the throughput over a TCP naïve to paths that modify window scaling.

HICCUPS helps advance the state-of-the-art in the field of detecting Internet packet header

modifications—currently an array of underused or impractical methods, all with some combination of deployment, incentive, or consistency issues that preclude integration into TCP. In doing so, HICCUPS satisfies a set of key design points that enable it to become part of TCP and answer the question of remote endpoint misinterpretation. As a demonstration of its capabilities, HICCUPS acted as the first known solution to capture measurements of packet header modifications within TCP and expose the results directly through the operating system itself. For example, the servers in our measurement infrastructure did not run any specialized server applications and instead ran standard HTTP daemons.

Beyond improving TCP performance, widespread HICCUPS deployment would provide invaluable data to researchers, policy makers, and protocol designers. Such data could help influence new protocol designs and facilitate the safe deployment of new and experimental protocol options. Measurements from running HICCUPS across a distributed and diverse set of paths discover a wide variety of (sometimes asymmetric) behaviors, including paths that modify, delete, or insert: sequence numbers, IPID or receive window, ECN, MSS, SACK permitted, timestamps, window scaling, Multipath TCP, and experimental options. Crucially, header modification behaviors are discovered by a HICCUPS-enabled TCP without prior application-layer coordination from the remote endpoint. Such a usage model also enables new diagnostic capabilities for network operators to help troubleshoot middlebox configurations on both the forward and reverse data planes.

9.1 Future Work: HICCUPS Protocol

Our promising initial work suggests several avenues in which the HICCUPS protocol itself could be extended or improved:

- **Deeper integration with TCP**: HICCUPS is the first solution to provide TCP with the ability to reason about the manner in which it is being interpreted by a remote endpoint. HICCUPS enables performance gains by selectively toggling certain options and extensions that are problematic due to a misinterpretation by a middlebox along a path. In its current state, however, our HICCUPS implementation does not automate those performance gains. For instance, we have not yet implemented the logic to automate the disabling of window scaling when a modification is detected. Further experimentation is required to fully characterize the path conditions where

TCP would be best suited by adjusting its default behavior.

- **Examination of other protocols beyond TCP**: Modifications to control information of other protocols such as UDP and IPv6 may help improve the inferences HICCUPS provides to TCP or help a system's networking stack improve performance for those protocols. UDP is extremely limited in terms of flexibility. A future design could leverage the only controllable value available within UDP, the ephemeral port. IPv6 is significantly more flexible due to its inclusion of extension headers enabled by the "Next Header" field, but further measurement would be required to determine if middleboxes generally support unknown extension headers. Use of a new extension header or the overloading of a current header could enable a stronger and more feature-rich version of HICCUPS for TCP over IPv6.
- **More coverage types**: Our current HICCUPS implementation provides 16 possible sets of fields that can be examined in a single connection. While no more bits are available in the 4-bit coverage transmission sections of the forward path HICCUPS integrity fields, it is possible to redefine the 4-bit value to reference a group of coverage types rather than a single coverage type. For instance, a coverage value of `0b0000` currently tells HICCUPS systems to use the `HNONAT` type, but it could instead point to a group of four different types. The receiving endpoint would then brute-force the four possible types to find the correct one, pending the presence of middlebox modifications. With 2^4 coverage type transmission space, and bins of size $n = 4$, HICCUPS could leverage 64 possible coverages. Not only could types be reserved for future extensions, current field granularity could be increased and new search strategies could be created.
- **Pluggable search strategies**: Since the efficient search strategies discussed in Section 5.6 do not need to be standardized to use HICCUPS, a system could potentially switch between different strategies, depending on which is best for a given path. A hot-swappable, modularized facility (similar to that of congestion control in Linux [112]) could be created for HICCUPS state maintenance and coverage selection algorithms.
- **Protect the entire connection**: Protection of the TCP 3WHS represents an important first step toward answering the question of misinterpretation since many extensions are negotiated there that affect the remainder of the connection. However, some

header modifications may only take place at later points during the connection (e.g., if congestion occurs and a packet is marked with the ECN CE codepoint). A design is presented in Appendix A.4 that has the potential to be used to protect all packets of a connection. If the initial HICCUPS check over the 3WHS determines that IPID and IP DF are maintained along the path, then an integrity value could be written into the IPID of each packet of the remainder of a connection. The hash values would be much smaller, perhaps requiring a probabilistic approach to deal with collisions.

- **Ability to leverage pre-shared secrets**: With HICCUPS, we assume the general (and more practical and useful) case of a pair of anonymous hosts communicating across the Internet. If a pair of hosts has some pre-shared secret at their disposal, usage of the ephemeral secret is not required to protect HICCUPS integrity and the pre-shared secret could be used instead. The integrity could be encoded and decoded by the endpoints immediately from the start of the connection. Further, the entropy of the TCP ISN would be fully maximized since it would also be encoded with the pre-shared secret. Such scenarios where a pre-shared secret might exist include: an IPsec environment, a pair of hosts that already use TCP-AO, or a coordinated debugging task. The implementation of HICCUPS would need to be altered (possibly by creating a new `setsockopt()` call) to allow applications, users, or key distribution systems to provide a key to protect HICCUPS integrity values.
- **Forward error correction**: Our idealized vision of HICCUPS (which would perhaps require a clean-slate approach to accomplish) is that a pair of TCP endpoints be able to automatically recover from a disruptive middlebox header modification without requiring any additional packet transit. Provision of the data necessary for such forward error correction (FEC) of header modifications is currently not possible within the small set of fields used by HICCUPS. In the event that HICCUPS no longer needs to operate under such tightly constrained space requirements (e.g., if other protocols are used), it is worth investigating coding techniques that would allow for FEC of packet control-plane information.

9.2 Future Work: Applying HICCUPS

We present the following areas in which HICCUPS inspires an avenue worth exploration in future application and usage scenarios:

- **Continued measurement studies**: We wish to continue our survey of Internet paths, analyzing header modifications in the wild and their impact on performance. While our measurements surveyed paths across a diverse set of countries and ASes, many were over university networks intended for networks research. We would like to broaden the collection of measurements to include users of residential and small business networks. To accomplish this work, we further plan to make our implementations available on our website [114] and to invite the community to make use of both the kernel and user-space versions of HICCUPS. We have already completed efforts to make the user-space version operate on all common platforms (i.e., Windows, Mac OS, Linux, and BSD). Greater variety in our measurements would further help to refine the implementations and the design of the HICCUPS protocol itself.
- **Usage by applications**: By providing an interface to user-space in the kernel implementation (Section 7.2), we make it possible for applications to control and process information from HICCUPS. In making this design decision, we envision a multitude of possible usage scenarios, including: use by network debugging suites such as Netalyzr [12], use by network scanning tools such as Nmap and nping [95] to attempt to model middlebox behaviors and fingerprint devices based on those behaviors (e.g., TCP `NOP` options that are not required for alignment), or use by a peer-to-peer application concerned about privacy or network neutrality issues.
- **Path influence**: We envision that, in the future, information resulting from HICCUPS checks could be used in a variety of ways to influence the path taken by traffic. Hosts and segments capable of source routing [31] could try to avoid paths containing a problematic middlebox. Other methods such as Multipath TCP could also be used to swap a connection over to a path that is more friendly to a new option or protocol extension. Also, networks themselves (e.g., content delivery networks (CDNs)) could incorporate HICCUPS results into their routing decisions. Routing decisions of CDNs are often proprietary, but it is well-known that they strive to find high-quality paths for traffic [121]. Existence or lack of existence of certain header modifications could be used to direct traffic through various network paths. Since most CDN routing is based on UDP traffic, more investigation into other protocols (as discussed in Section 9.1) would be necessary.
- **Deployment in other domains**: Military and industrial networks often operate un-

der a much different set of guidelines than typical Internet-connected subnets. In particular, some standalone or "airgapped" networks may operate in complete autonomy or even be connected externally at specific times (e.g., when a ship is at port). Future work should explore the potential for HICCUPS to aid in quickly debugging and understanding the impact of opaque middlebox deployments on such networks. Portable network "test kits" including HICCUPS capabilities could be deployed to operators of these networks to examine path properties both internally or when they route through a new interconnection. Such an exploration would likely provide opportunities for technology transfer of HICCUPS beyond the effort to integrate it into networked operating system stacks.

- **Root cause of issues**: While HICCUPS can provide information about which header fields are modified along which paths, it does not provide any insight as to the reason for that modification. For instance, HICCUPS cannot differentiate between accidental modifications to window scaling due to a misconfiguration or purposeful efforts to artificially flow control a connection. In order to better understand and characterize these differences, HICCUPS measurements could be accompanied by a range of other probing techniques that detect more high-level devices such as TCP-terminating proxies and load balancers. The measurements should be synthesized to extract high-level information that the research community can use to reason about typical causes for performance-penalizing middlebox behaviors.
- **Is the path changing?**: Another interesting question to ask about the asymmetric path behaviors we detected in the measurements in Chapter 8 is: *to what degree are those asymmetric paths versus asymmetric modification behaviors by a middlebox?* In other words, are we actually measuring two different paths during a single 3WHS exchange. In order to better answer such a question, advanced traceroute [38] data needs to be captured in parallel with HICCUPS measurements and examined for path differences. A tool such as Tracebox [39] could also be used to correlate HICCUPS results.
- **HICCUPS-enabled middleboxes**: Under plain HICCUPS (Chapter 5), middleboxes and intermediary devices would be able to extract and analyze the integrity information if they desire. This prospect begs the question: *how can they do so safely?* Suppose that a middlebox close to the client wishes to add HICCUPS integrity to

outgoing connections before they traverse the Internet, that system would first need to ensure that integrity is not already present or overwritten. Otherwise, overwriting the values to add HICCUPS would induce a false inference. Perhaps HICCUPS state could be combined with an optional middlebox discovery option [82] for safe deployment. In particular, mobile networks have a high presence of intermediate devices to assist the network, and the range of low-powered devices it hosts [122], which could attempt to make use of HICCUPS integrity information.

- **Middlebox behaviors over time**: HICCUPS could play an integral role in measurement studies designed to determine whether middleboxes are improving or harming network performance overall. As shown in this dissertation, numerous performance-impacting issues can and have occurred due to the addition of intelligence to the network. Often, the intelligence is added in an effort to aid either security or performance. Such a measurement study could correlate HICCUPS-detected header modifications with overall end host performance and help to answer the question, *is the increased presence of middleboxes helping or hurting performance?*

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Design Catalog

In the process of arriving at our final design for HICCUPS detailed in Chapters 5 and 6, we created an array of potential designs for both the transmitting and protecting of integrity values. In this appendix, we document each of those designs so that they will be available for reference in the event they are needed by future work. For instance, the variant listed in Section A.4 may one day be applied as an extension to HICCUPS that could be used to protect the entire connection after the 3WHS. Each variant comprises a separate section within this appendix and is closed with a page break to aid readability.

A.1 TCP design variants

For each subsequent variant section, we define the following key properties:

Throughout connection

We can either protect just the 3-way handshake or the entire connection. Detecting some modifications requires examining a full connection. For example:

- A middlebox, *M*, modifies initial TCP sequence numbers, but fails to also update SACK blocks into new sequence number window. The SACK blocks will not appear until after the handshake.
- *M* shrinks the TCP receive window in order to throttle the connection between *A* and *B*. This could happen at any point during the connection.

Diagnostic mode

Some of the variants are not able to coincide with a connection for an application (e.g., www, ssh, etc.). If so, they will be marked as "diagnostic only," meaning that no application data should be sent through the connections used by that variant. It can still operate with any open TCP port on a server, but the connection that is created only serves the purpose of checking packet header integrity, after which it is closed.

The ability for a pair of endpoints to create a diagnostic connection creates further issues:

- How does each endpoint detect that a given connection is diagnostic?

- How do you prevent *M* from determining the same thing? If *M* knows which connections are diagnostic, it can adjust its behavior for only those connections.

Fields used

This item lists the fields within the TCP or IP headers that each variant uses in the transmission of integrity or status information.

Raises bar on *M*

At this point we are only focused on communicating integrity information. Chapter 6 takes a deeper look at protecting the integrity and what can be done to stop or at least discourage the information from being modified by the middleboxes we target. For purposes of comparison with the design variants in that chapter, we include a quantification of the level of protection afforded by each variant.

A.1.1 Notation

The diagrams of each variant within this document will show information passing between two hosts, *A* and *B*. The information itself will be consistently described in the format shown in Figure 1.

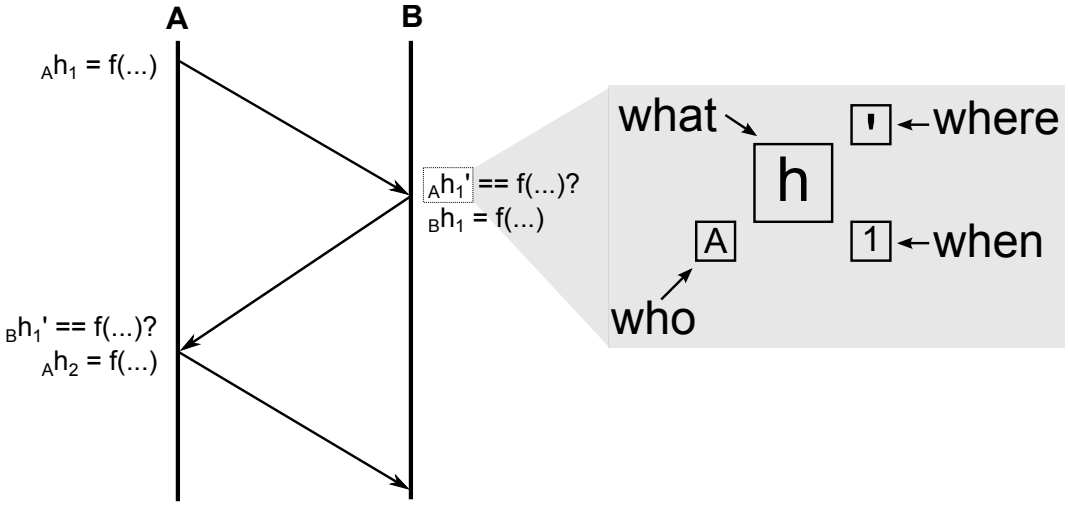

Figure 1: Standard notation for variant diagrams

What: The type of information (e.g., *salt*, *h*, etc.)

Who: The party that originated the information (either *A* or *B*)

Where: A prime symbol ($\prime$) here indicates that this is the value of the information after having transited the network. In other words, this information may have been modified by M.

When: A number n to indicate that this is the n^{th} piece of information of similar type from the same origin. If not present, then it means the information is only sent once from that origin.

Also discussed is $f()$, a publicly known hashing function that converts field state representations to hash values.

A.2 Variant 1: Opportunistic HICCUPS

In this variant, *A* and *B* each embed a hash and salt value in their SYN and SYN-ACK, respectively.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN)
	IPID (on first packet)
Raises Bar on *M*:	Not really. *M* must recalculate two hash values and at most store one packet header for up to half of an RTT.

A.2.1 Detailed Description

The opportunistic variant of HICCUPS has the ability to inform both parties in a TCP connection if their packets were unmodified, without requiring a special diagnostic connection or an extra RTT. This is important for high-performance applications that cannot afford any added delays. The packets exchanged as part of this check look no different to the network than any other similar packets, the only difference is that they have an ISN and IPID that have special meaning. The opportunistic check was designed to operate as part of a normal connection between two hosts that may or may not be using HICCUPS.

An example timeline of the opportunistic check between two hosts *A* and *B* is shown in Figure 2, where *A* initiates the TCP active open. When *A* sends the first packet, it must include a random string (the salt) and the result of the function of that packet's fields and salt value. The salt value is placed in the IPID field and the output from the function is placed in the ISN.

We define the following:

$$\begin{aligned} {}_{A}salt &\leftarrow rand() \\ {}_{A}h &= f(fields_{\texttt{SYN}}, {}_{A}salt) \end{aligned}$$

The function $f()$ is public (e.g., a known hash function). This allows the host at the other end of the connection to compute ${}_{A}h'$ using the standardized function and the fields and salt

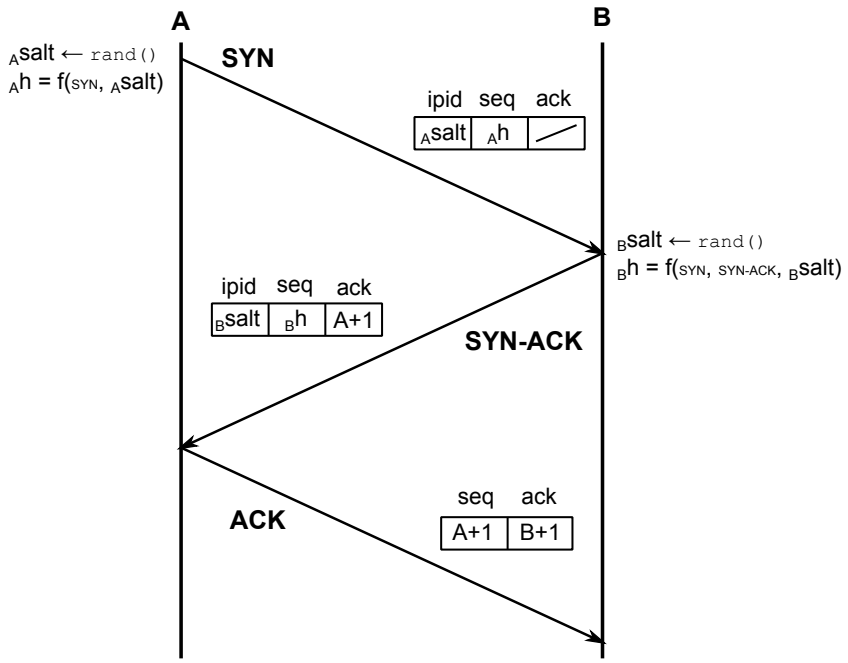

Figure 2: Timing diagram with no modifications

values of the packet from A as seen by B. If there were no modifications, then ${}_Ah' == {}_Ah$. Should they match, B can say that A's packet was unmodified.

At this point, B generates its own salt and ISN value for the returning SYN-ACK packet. If the checks by B pass, then it should incorporate a way for A to know that they passed as well. This can be done by including something known to A in the function input. The fields from the SYN packet can be combined with the fields from the SYN-ACK packet in calculation of the sequence number used in the SYN-ACK:

$$\begin{aligned} {}_Bsalt &\leftarrow rand() \\ {}_Bh &= f(fields_{\texttt{SYN}}, fields_{\texttt{SYNACK}}, {}_Bsalt) \end{aligned}$$

Should the check fail at B, it could inform A by leaving out the $fields_{\texttt{SYN}}$ input from $f()$. This would yield the following instead:

$${}_Bh = f(fields_{\texttt{SYNACK}}, {}_Bsalt)$$

When A receives the SYN-ACK reply from B, it must check the packet's sequence number against both possibilities. The function must be computed using each of the two input combinations B may have used above and determine whether either of them match the sequence number it sees. Table 1 lists all possible outcomes at each end of the exchange.

At host B after receiving SYN:	
${}_Ah == f(fields_{\texttt{SYN}}, {}_Asalt)$	SYN unmodified
$else$	SYN modified or A not capable
At host A after receiving SYN-ACK:	
${}_Bh ==$ $f(fields_{\texttt{SYN}}, fields_{\texttt{SYNACK}}, {}_Bsalt)$	SYN and SYN-ACK unmodified
${}_Bh == f(fields_{\texttt{SYNACK}}, {}_Bsalt)$	SYN modified but SYN-ACK not
$else$	SYN-ACK modified or B not capable

Table 1: Possible outcomes of the opportunistic check

A.2.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must simply recalculate ${}_Ah$ and ${}_Bh$ after performing its modifications. M does not need to regenerate salt values; it can reuse the ones chosen by A and B. Finally, M must be able to store the SYN fields until it can calculate ${}_Bh$. At most, this will be until it sees the SYN-ACK return from B. Figure 3 summarizes this process.

A.2.3 Pros

- Interoperable, incrementally deployable

A.2.4 Cons

- Does not protect the entire connection
- Can be disabled by overwriting initial sequence numbers
- Bob cannot distinguish between Alice not being HICCUPS-capable and a modified field

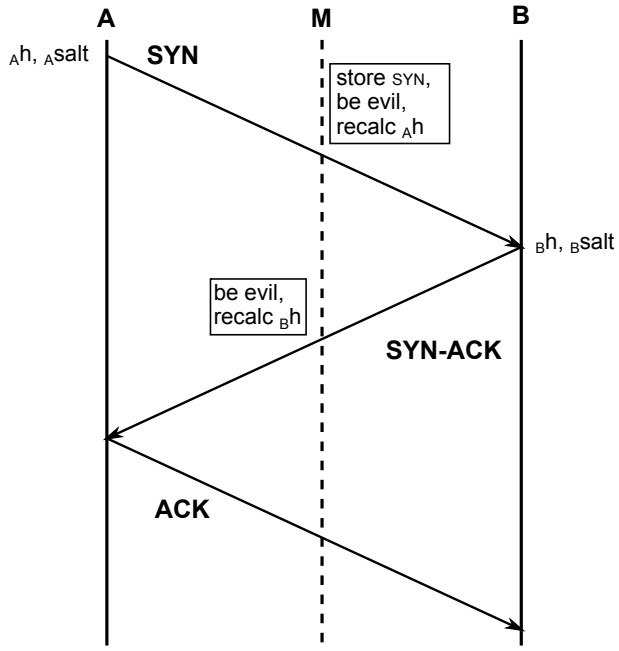

Figure 3: Necessary actions to fool A and B

A.2.5 Thoughts and Status

This variant is simple and easy to understand and implement. It represents a promising approach to achieving good compatibility properties. Implementations of this variant currently exist as the basis for HICCUPS as presented in Chapter 5. Both a kernel and a user-space version are available. See Chapter 7 for more details.

A.3 Variant 2: Offset Sequence Numbers

In this variant, hash values are added to the sequence numbers in each direction.

Throughout Connection:	Yes
Diagnostic Mode:	Yes, diagnostic only
	Mode Hidden? Reasonably so
Fields Used:	SEQ and ACK
Raises Bar on *M*:	Not really. *M* has to guess that the connection is in diagnostic mode, recalculate two hash values, and store a sequence number for up to half of an RTT.

A.3.1 Detailed Description

This variant performs additive increases to the sequence number of each packet in the connection, with that increase being expected by the other HICCUPS-capable end-host of the connection. The result of the public function, $f()$, is added to the sequence number at the completion of the handshake. This new sequence number will be outside the receive window of the opposite host, forcing a duplicate ACK from a non-HICCUPS host.

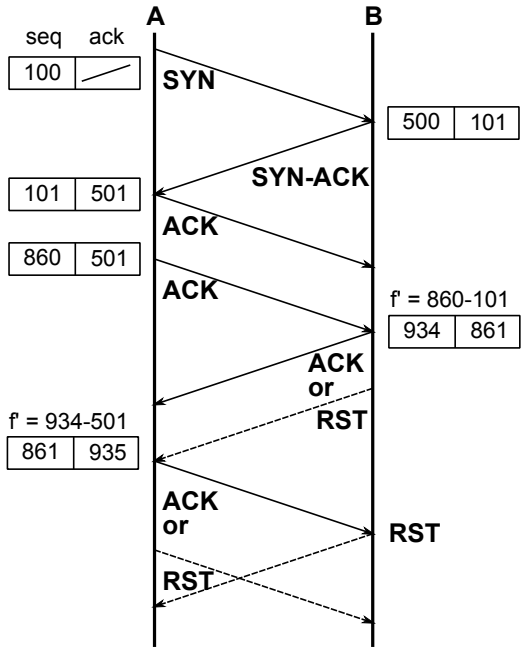

Figure 4: Two HICCUPS hosts (no mods)

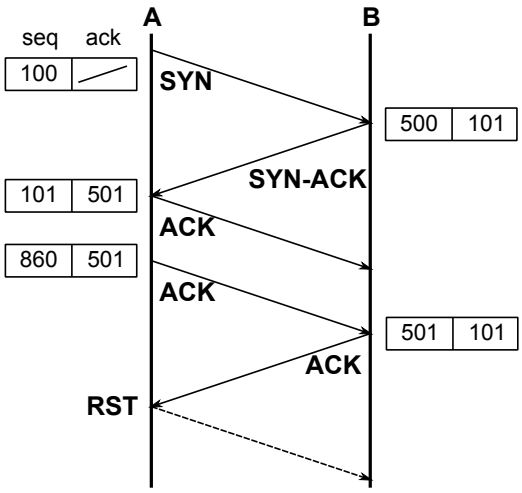

Figure 5: B not HICCUPS host

The example timeline in Figure 4 is a demonstration of the offset sequence number variant as carried out between two HICCUPS-capable hosts. After *A* completes the handshake, it immediately sends the next packet where the sequence number is $f()$ more than before. In this example, $f()$ was 759. Upon receipt, *B* can recompute $f()$ from the values of the packet and check to see if it matches the difference in sequence numbers. If it does, *B* does the same thing as *A* and sends a packet with an additively increased sequence number. If it does not, then *B* can send a RST packet to end the connection. *A* then does a similar check in response.

Figure 5 shows the timeline where *B* is not using HICCUPS. If *B* replies with a DupACK, we know it must be plain TCP and does not understand our offset sequence number. Thus, we cannot get an integrity check out of this host. Something to note here is that some systems will not respond with a DupACK unless the difference in sequence number is greater than the host's receive window. This can be handled by adding a value to $f()$.

The primary benefit of this variant is that it enables each end of the connection to distinguish between a failed integrity check and a host not using HICCUPS. This eliminates the ambiguity present in the opportunistic variant from Section A.2. In order to accomplish this, the connection is completely used as a diagnostic connection that does not handle any application data. It should be possible, however, to begin a connection using the opportunistic check and then switch to this variant if there was an issue with the results of the first check.

A.3.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that no modifications were made, it must first recognize that a diagnostic connection is taking place and then overwrite the sequence numbers with adjusted values.

Recognization of the mode is a tricky issue. The packets will appear to be out-of-order to most systems, and the check must match up to detect the mode. Granted, *M* could match up the check the same as *B* can, but either way, we introduce a little probability into *M*'s decision.

Once *M* makes the decision that a given connection is performing the offset sequence num-

bers check, it must perform the actions as shown in Figure 6 in order to fool A and B into thinking that their connection has integrity. This includes saving a 32-bit sequence number for up to half of an RTT and recalculating two hashes.

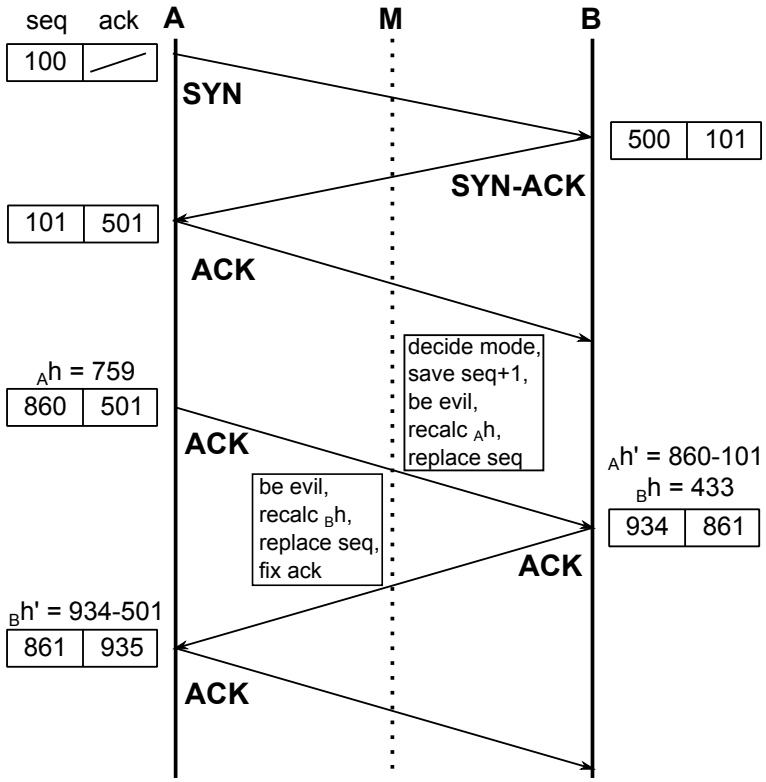

Figure 6: Necessary actions to fool A and B

A.3.3 Pros

- Don't change ISNs, so don't need salt in IPID
- Should get through systems that securely randomize initial sequence numbers since we only care about the deltas

A.3.4 Cons

- Essentially injecting junk into the network which may have unintended consequences
- Some systems may react poorly to the out-of-window sequence number

A.3.5 Thoughts and Status

One troubling aspect of this variant is that we do not know what to expect from the network as a whole in handling the offset sequence numbers. This variant essentially involves injecting "junk" packets into the network and assuming (likely incorrectly) that they will be transited properly by the network. This variant is probably not useful on its own.

However, a really large positive aspect of this design is that it provides a way of successfully distinguishing between a HICCUPS host and a non-HICCUPS host without marking up the packets in a manner that may introduce incompatibilities.

A.4 Variant 3: Probabilistic Hashes

In this variant, smaller hashes bounce back and forth between A and B. The result is probabilistic over many trials throughout the connection.

Throughout Connection:	Yes
Diagnostic Mode:	None
Fields Used:	IPID
Raises Bar on M:	No. Per packet it sees, M only has to recalculate one hash and store one hash for up to half an RTT.

A.4.1 Detailed Description

If we allow the hashes to be really small (for instance a single byte), we can squeeze two of them into the IPID fields of every packet in a connection. Obviously with such a small range of outputs for $f()$ we should expect a fair amount of collisions. However, if we do these checks on each packet over the life of a connection, the probability of all of them being collisions becomes very small. This is akin to the way the ECN nonce [48] works.

The timing diagram in Figure 7 shows the small hashes bouncing back and forth between A and B. For the purposes of this example, imagine that the IPID field is split in two and the upper byte is used for A's hashes with the lower byte used for B's hashes. The host sending a packet includes a mini hash of its fields in its half of the IPID. Also, for all but the very first packet, the other end host's hash can be echoed back to them. This gives both ends of a connection visibility into modifications in each direction.

Upon receipt of a packet, the receiving end host can perform two checks:

- Does the echoed hash equal what was sent?
- Does the hash from the other end equal a hash of the fields?

The results of these two checks give the host insight over modifications to packets sent and received, respectively.

Due to the short length of the hashes, there is a larger chance of having collisions. This means that there is a sizable chance our check may not work, even with M being nice and not altering fields. We can tolerate this, though, because each packet is another trial and if

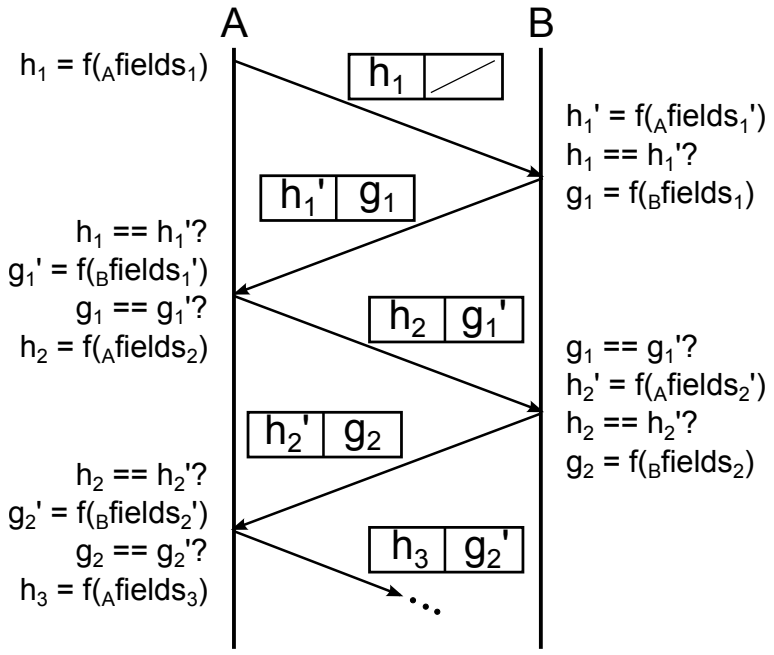

Figure 7: Timing diagram with no modifications

M is modifying fields (but not necessarily trying to fool us), we are bound to see it with the majority of the packets.

Note that we make a small adjustment to the notation from Section A.1.1. We substitute h for ${}_{A}h$ and g for ${}_{B}h$.

A.4.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that no modifications were made, it must replace each of the mini hashes exchanged between them. For each packet that passes through *M*, it must recalculate a hash and store the original for up to half an RTT.

A.4.3 Pros

- Simple, easy to understand
- Lightweight, relies on probabilities over multiple trials
- Hash is echoed back to you as the other end saw it

A.4.4 Cons

- Need to use on lengthier connections

- Results may be fuzzy
- Easy for *M* to intercede

A.4.5 Thoughts and Status

This variant seems like it would work well, unless there is in-network fragmentation. Also, similar to the opportunistic HICCUPS approach, once middleboxes become aware of the technique, the integrity values may become less trustworthy as it will be very easy for a middlebox to recompute the hashes. *M* has to do about half of the sum of the work *A* and *B* must do in order to fool them.

A.5 Variant 4: Hash Striping with Resets

This is an improved variant that transmits the integrity hashes in triplicate in order to withstand a modification to one of the integrity fields. Having three-way striping of the hash gives reasonable proof that HICCUPS is used by the SYN initiator and allows for a TCP RST to be sent when the hashes fail to match.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN) IPID (on first packet) TCP Receive Window (on first packet)
Raises Bar on *M*:	Not really. *M* must recalculate two hash values and at most store one packet header for up to half of an RTT.

A.5.1 Detailed Description

The impetus for this variant came after performing initial tests on PlanetLab using the standard opportunistic HICCUPS. The results are discussed in more detail in Chapter 8, but about 13 percent of the nodes we tested experienced some combination of either ISN translation or modification of the IPID field. While it was good that we were able to detect that a packet header modification was taking place, we lose visibility to any other changes made to the packet by *M*. This is because our integrity hash is overwritten and we lose the ability to check the smaller subsets of header fields that do not contain the ISN or IPID fields.

This variant solves the issue by copying the integrity hash into three separate fields of the packet header. In addition to the fields we used before, ISN and IPID, we also use the TCP receive window. It was realized that the value of this field is not important during the three-way handshake and could be repurposed. A random salt value is still required in order to ensure proper randomization of the ISN, so all hashes are set at 16 bits in length and the 16-bit salt is placed in the upper half of the ISN with the hash going into the lower half. The layout within the fields is shown below in Figure 8.

The host that receives the SYN will check to see if any two of the three hash bit ranges

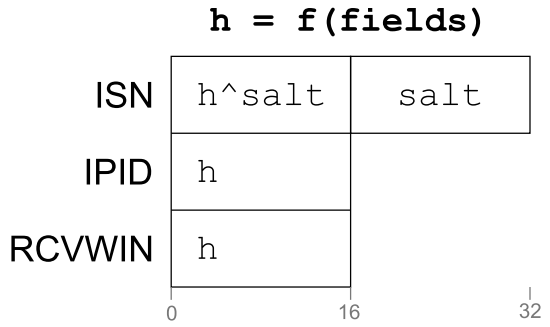

Figure 8: Hash and salt layout in header fields

match. This will allow transmission of the integrity hash even if one of the three fields were modified by M. A "majority rules" vote is taken from the three fields and that hash is assumed by the receiver to be the HICCUPS hash sent by the SYN initiator. Obviously, if any two of the fields are modified, we lose granularity and can only tell that the path integrity has failed.

A key observation is that it is highly unlikely that any two of the 16-bit fields would be exactly the same unless they were originally set that way by a HICCUPS-enabled SYN initiator. In the worst-case, a randomly set ISN will match either the IPID or receive window value, causing the remote end to infer a HICCUPS capability and calculate path integrity, which will fail. Because of this unlikelihood, we extend this variant with a TCP RST to enable a feedback mechanism.

When the SYN receiver detects a HICCUPS hash (by finding two of the three hash fields with the same value), and then determines that hash to fail the integrity check, it will respond with a TCP Reset packet. This RST will act as feedback to the SYN initiator that bits were modified while the SYN was in transit to the receiver. It can be differentiated from a RST due to a closed TCP port by sending a SYN without any HICCUPS hashes. In this case, the receiver will not find two of three fields with the same hash and must not respond with a RST since the SYN initiator is assumed to be not HICCUPS-capable.

If the hashes both exist and pass integrity checks, a similar layout is used to transmit integrity in the SYN-ACK. An example transaction is shown in the timing diagram in Figure 9.

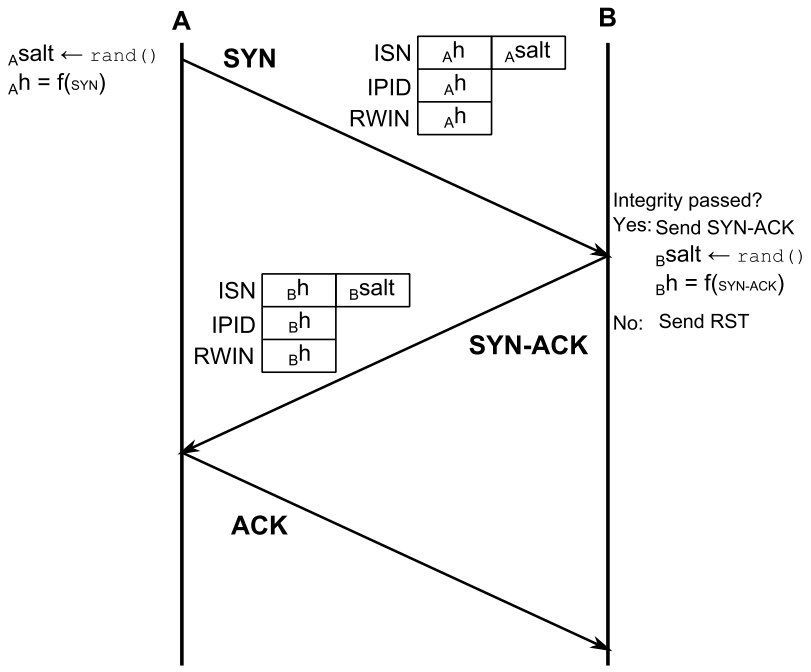

Figure 9: Timing diagram with no modifications

All other aspects of this variant are similar to the Opportunistic variant in Section A.2. Both parties are informed about the path integrity and it fits with the TCP handshake, so only a single RTT is required for integrity status to be obtained. The only downside is that we now use smaller hashes, 16 bits long instead of 32.

A.5.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must simply recalculate ${}_Ah$ and ${}_Bh$ after performing its modifications. M does not need to regenerate salt values; it can reuse the ones chosen by A and B. Finally, M must be able to store the SYN fields until it can calculate ${}_Bh$. At most, this will be until it sees the SYN-ACK return from B. This is exactly the same as for the Opportunistic variant, but instead the hash has to be written three times.

One possible weakness of this variant (depending on the viewpoint) is that, along with the receiver, any middleboxes along the path of the packet can tell that the SYN initiator is using HICCUPS. This saves a devious middlebox from having to overwrite hashes on all SYNs it sees and instead just focus on ones where two out of the three fields have the same

value.

A.5.3 Pros

- Interoperable, incrementally deployable
- Withstand modifications to any one of the three fields used to transmit integrity
- Gives status feedback through RST packet (stopping the connection before it starts and allowing the initiator to retry with less features enabled)
- Can distinguish between HICCUPS-capable and a failed integrity check

A.5.4 Cons

- Does not protect the entire connection
- Still breaks if any two integrity transmission fields are modified
- Uses smaller length hashes and would be prone to more collisions

A.5.5 Thoughts and Status

This variant is fairly simple to understand, but implementing the RST work may be difficult in kernel. The variant's key features of feedback and hash modification tolerance are definitely needed after seeing PlanetLab results. RSTs can make things messy for non-HICCUPS hosts, however, and if we do not do the RST we are unsure of how else we can transmit the status feedback.

A.6 Variant 5: Hash Rainbow

This variant is similar to the previous variant in Section A.5, except that the TCP RST is not used as feedback response. Instead, four bits are taken from each hash and used to carry the status. To avoid collisions while allowing for smaller hashes, a different hash function is used for each of the three hashes.

Throughout Connection:	No, handshake only
Diagnostic Mode:	None
Fields Used:	Initial Sequence Number (ISN) IPID (on first packet) TCP Receive Window (on first packet)
Raises Bar on *M*:	Not really. *M* must recalculate six hash values and at most store one packet header for up to half of an RTT.

A.6.1 Detailed Description

This variant transmits an integrity representation in three places, the ISN, IPID, and TCP receive window. For each of the three fields, the integrity input is hashed using one of three different hashing functions. For example, the hash we place in the ISN may use MD5, while the hash we place in the IPID uses SHA-1 and the hash in the receive window uses SHA-256. The layout is described in Figure 10.

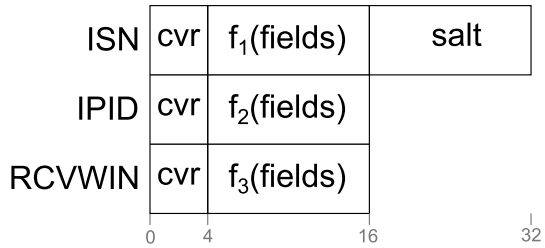

Figure 10: Hash and salt layout in header fields

The reason behind this "rainbow" of hashes is that the hash values themselves are only going to be 12 bits long. That means there is a 2^{-12}, or $\frac{1}{4096}$, probability that a random number would be misinterpreted as a valid hash showing correct integrity. Since this probability is fairly high, the multiple hashing functions are used to reduce the chance of a false positive. The chance that the values placed in any two fields by a non-HICCUPS sender would

match the expected outputs of two different hashing functions should be much lower.

Since the hashes have been reduced in length to 12 bits, that leaves four bits to be used for transmitting status information. In the SYN, these four bits carry the coverage type that the SYN initiator would like for the SYN receiver to use when it builds the integrity in the SYN-ACK. On the returning SYN-ACK, the four bits carry the status of the SYN integrity. The transaction is summarized in Figure 11.

For the status bits on the SYN-ACK, the lowest order bit is used to signify whether the hash in the RCVWIN field of the SYN matched. The next lowest bit signifies a match in the IPID hash, and the third bit signifies a match in the ISN hash. The highest of the four status bits is always set to a value of one so that the TCP receive window value will not go lower than 32,000.

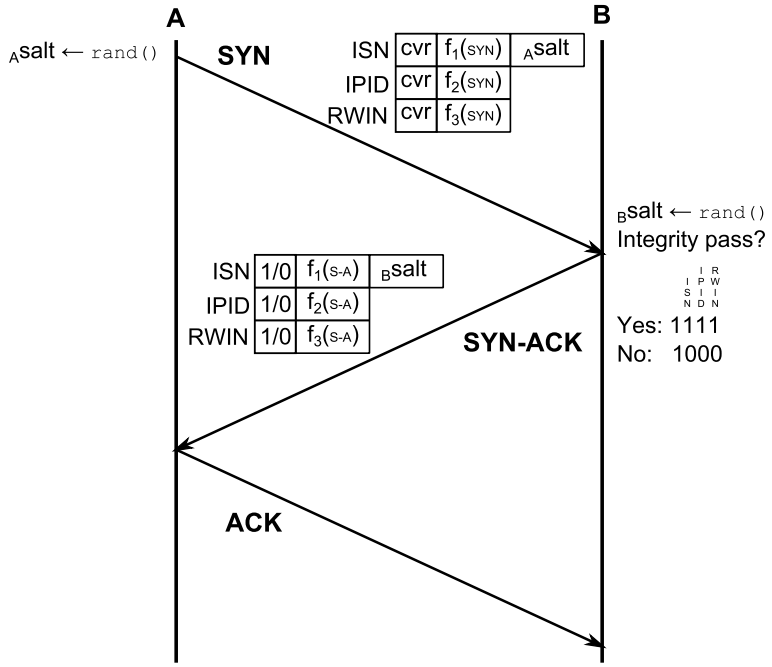

Figure 11: Timing diagram with no modifications

A.6.2 Faking Integrity

The actions that need to be taken by a middlebox and the issues involved are the same as with the variant in Section A.5. The only difference is that, due to the different hash functions, middleboxes can no longer immediately tell that a packet is HICCUPS-enabled.

This fact forces a devious middlebox to overwrite hashes on all packets if it wants to fake integrity. The middlebox will also have to perform all three different hashing functions for each packet it modifies.

A.6.3 Pros

- Interoperable, incrementally deployable
- Withstand modifications to any one of the three fields used to transmit integrity
- Gives status feedback
- Won't disrupt any connection attempts due to RST

A.6.4 Cons

- Does not protect the entire connection
- Still breaks if any two integrity transmission fields are modified
- Uses smaller length hashes and would be prone to more collisions (but is helped out by the three different hash functions)
- Can't distinguish between non-HICCUPS capable and failed integrity check (but at least the middlebox can't either)

A.6.5 Thoughts and Status

This variant seems like the best option. Our primary concern with this strategy is the small size of the hashes. Validation was required to quantify the performance of the rainbow of hash functions at reducing collisions in the hashes.

This variant has many good qualities, listed above in the "Pros" section. The same combination of good qualities is not present in any of the other variants, making this variant very enticing, but the concern of the small hash sizes must be must be managed. We took steps to control this by the choice and combination of hashing algorithms in our implementation.

A.7 Variant 6: CoinFlips

In this variant, a coin flip is added to the probabilistic variant from Section A.4 to try to raise the bar on M.

Throughout Connection:	Yes
Diagnostic Mode:	None
Fields Used:	IPID
Raises Bar on M:	Yes. M must do at least as much work as either endpoint. Each RTT it must calculate five hashes and store three for up to half an RTT.

A.7.1 Detailed Description

At its core, this variant is the same as the probabilistic variant described in Section A.4. Except now, we have added a bit of randomness to how each side encodes the hash it echoes. This forces M to do some calculations to determine the result of the coin flip so that it knows how to encode the echo hash on the return packet.

In the conversation shown in Figure 12, A initiates the active open and B handles the coin flips. It is A's job to determine the value of the flip and use it to properly encode the hash it is about to echo back to B. B then checks to ensure that the echoed hash was encoded with the same side of the coin that it used.

For the notation, we make the same adjustment as the last section where we substitute h for ${}_{A}h$ and g for ${}_{B}h$. We use the pre-subscript to denote the value of the coin flip.

A.7.2 Faking Integrity

This variant includes lots of extra steps over the Passing Hashes variant, but it does raise the bar more on M. In order to force more work upon M, we must do a little more ourselves as well. The question is: did we make M do more extra work than we had to do? The conversation where M tries to fool is shown in Figure 13.

In order for M to fool A and B into thinking that no modifications were made, it must calculate five hashes per RTT and store three for up to half an RTT. This is a much greater burden on M than the "calculate one, store one" requirements of the Passing Hashes variant.

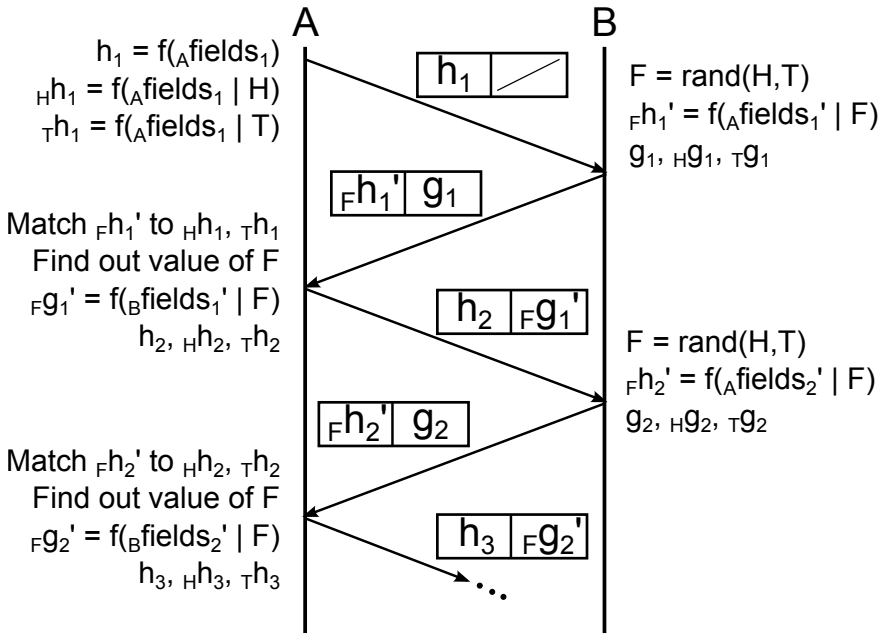

Figure 12: Timing diagram with no modifications

However, we have also increased the load on *A* and *B*. Per RTT, *A* and *B* must calculate up to four hashes each and store up to two hashes. The good thing though, is that we have raised the bar on *M* to just above the work required by either *A* or *B*. So in order for *M* to fool us, it must work harder than either end point.

A.7.3 Pros

- Raised the bar a bit on *M*, but not up to the sum of the work of *A* and *B*
- Has many of the same pros of the probabilistic variant

A.7.4 Cons

- More expensive than the probabilistic variant
- Still needs many trials (packets) to get a good reading

A.7.5 Thoughts and Status

This variant shows much promise, but does not quite deliver all that we desired: where *M* has to do as much work as *A* and *B* combined. We tested some other closely-related variants, but none seemed to get the ratio of *M*'s work to *A*'s work as high as CoinFlips did.

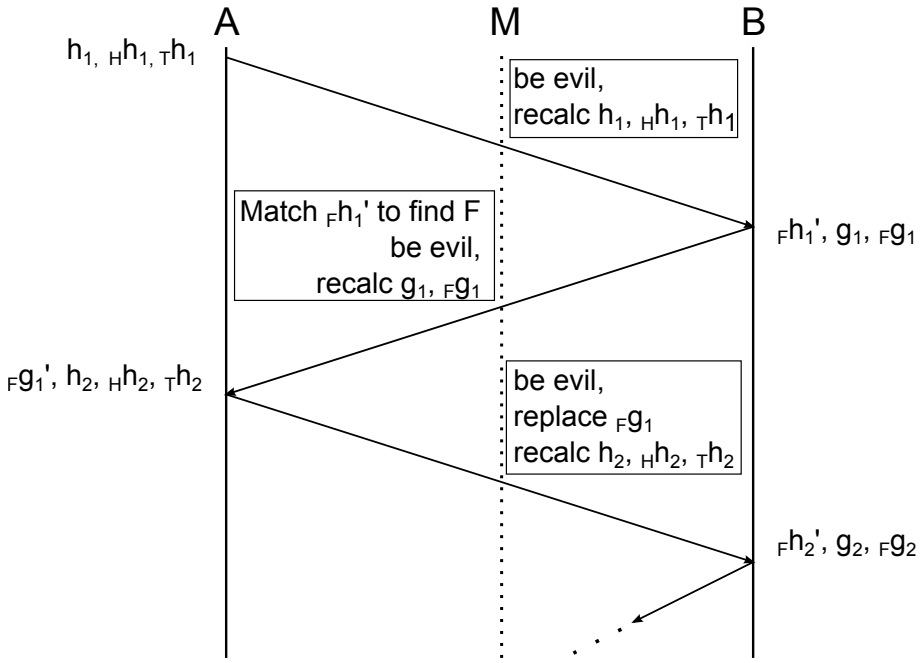

Figure 13: Necessary actions to fool A and B

A.8 Variant 7: HashCash

The goal of this variant is to stop a middlebox from easily overwriting fields by requiring the hashes to have a specific property.

Throughout Connection:	No, handshake only
Diagnostic Mode:	Yes, diagnostic only
	Mode Hidden? No, would be mostly detectable
Fields Used:	Initial Sequence Number (ISN)
	IPID
Raises Bar on *M*:	Yes. After modifying a packet, *M* must spend CPU cycles to find a good value of R if it intends to fool *A* and *B*.

A.8.1 Detailed Description

In this variant, we require the hashes to show that some computation work was accomplished by the originator. One such way we could do this is to require that the hashes end in a minimum number of zeros. In order to generate a hash that has this property, a system

must essentially brute force different values to include with the input to reach the desired property on the output. In our notation, we will call this special value R. When a valid R is given as input to the hashing function along with the field state representation, it should produce an output with at least the required number of zeros at the end.

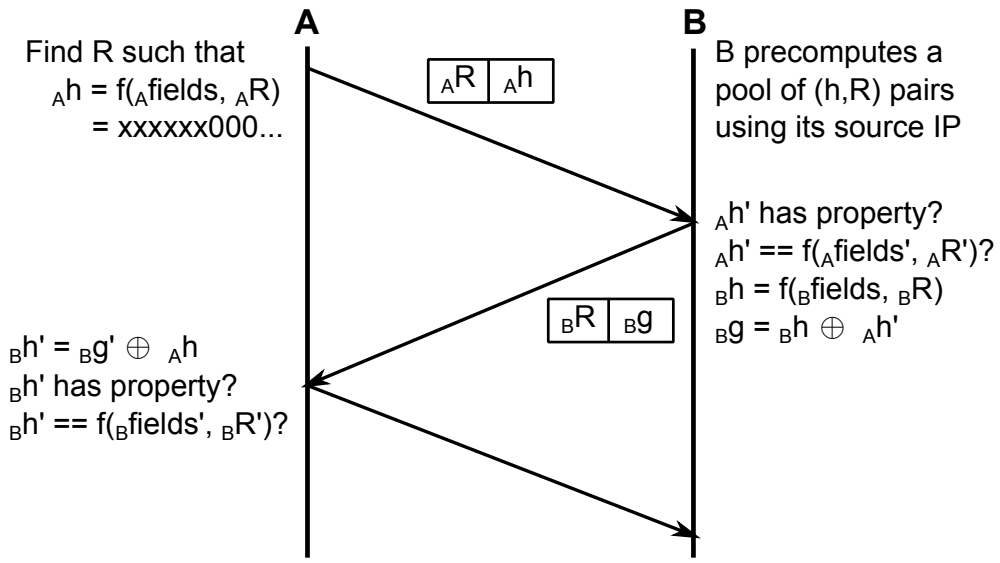

Figure 14: Timing diagram with no modifications

We then leverage the fact that M does not know when a given connection will start, nor what parameters it will have. This means that M cannot start working on the puzzle until it sees the SYN packet come through. The connection initiator, A, took some time to solve the puzzle before it sent the SYN. This means that there was some lag in starting the connection, but this is less of an issue if it is a diagnostic connection.

After A begins the connection by transmitting the (h,R) pair, B performs two checks on the pair:

1. Does h have the property (end with enough zeros)?
2. Does h equal a rehash of the packet's fields?

B then responds with an (h,R) pair of its own that it has precomputed with some common values and its source IP. We will have to specially craft the set of fields for this check so that B can perform this as precomputation.

A.8.2 Faking Integrity

In order for M to fool A and B into thinking that no modifications were made, it must quickly make its changes and calculate a new (h, R) pair as soon as it sees A initiate the active open. Depending on how difficult we tweak the puzzle, this should take a detectably long enough amount of time. M will have to delay the packet and we can look for the abnormally long RTT.

A.8.3 Pros

- Good at discouraging a middlebox from interfering

A.8.4 Cons

- Forces the endpoints to spend CPU cycles solving hash puzzles
- Lag time from when A's user requests a connection until it solves the puzzle and builds the SYN

A.8.5 Thoughts and Status

This is a big step forward over the previously discussed variants at raising the bar on the middlebox. Ultimately, we believe that the **cons** listed above would force this to be only used in a diagnostic mode connection. This gets into a question of can M tell whether we are in the diagnostic mode or not.

If M can easily detect when two hosts are in diagnostic mode, it can just play nice in those cases and change packets in all the rest of cases. In this variant, there is nothing to disguise the mode. If a middlebox sees hashes that do not satisfy the two checks, it can freely modify packets. Variants in subsequent sections try to tackle this problem.

A.9 Variant 8: Reverse Hash Chain

The salient feature of this variant is that it hides the existence of a check from *M* until the full hash chain is revealed. All of the chain's hashes look random until the salt is revealed.

Throughout Connection:	Yes
Diagnostic Mode:	Optional
	Mode Hidden? Yes, until chain revealed
Fields Used:	IPID
Raises Bar on *M*:	Yes for detection, but *M* can still easily overwrite chain. Can also make a strong argument using a random sampling of checks.

A.9.1 Detailed Description

This variant employs a reverse hash chain to obscure whether the hashes are a check or just random bits. Instead of directly embedding the hashes in the packet, we run it through the hashing function several more times and embed the final output. This results in a chain of values where it is really easy for a computer to determine a relationship in one direction, but not the other. By starting off the connection with the end of a hash chain, we make it very difficult for a middlebox (and the other endpoint) to trace the chain in reverse and determine the original value.

Figure 15 shows an example conversation using the reverse hash chain check. For illustration purposes, we fix the length of the chain at **four packets**, but it can be any preset length. The length can be tuned according to how long you want to delay detection. The final (in our case, fourth) packet of the chain reveals the information needed to reconstruct the chain. We call this random value the salt. The salt is needed because it prevents *M* from reconstructing the chain on the first packet.

The following order of events occurs for the length 4 chain:

1. *A* chooses a random salt value, c_0
2. *A* hashes (c_0, $fields_1$) to get c_1
3. *A* hashes c_1 to get c_2
4. *A* hashes c_2 to get c_3

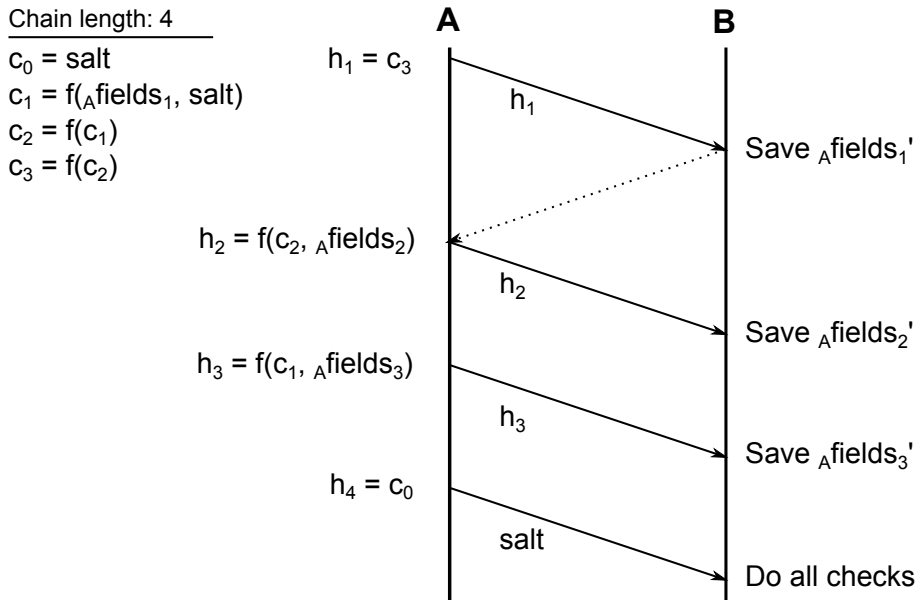

Figure 15: Timing diagram with no modifications

5. *A* embeds c_3 into first packet and sends it
6. *A* embeds the hash of (c_2, $fields_2$) into the second packet
7. *A* embeds the hash of (c_1, $fields_3$) into the third packet
8. *A* embeds the salt, c_0, into the fourth and final packet

Now *M* and *B* can both reconstruct the chain and verify the fields hashes. The extra hashes in steps five and six provide integrity over the middle packets of the chain. *B* can go back and check these too once it gets the salt.

The key effect we have had is that *M did not know until the end of the chain whether we were actually doing a check*. On the first packet, we force *M* to commit to either:

- overwriting the chain (which it can easily do), or
- leaving the hashes unmodified

If *M* always chooses to overwrite just to be safe, it will be doing more work than necessary since some connections will not use a check. Therefore, the burden on *M* is much greater than on the endpoints, since they only have to expend the hash chain computations when they decide to do a check. If *M* fails to overwrite the chain beginning with the first packet,

the connection will fail our checks and we can detect it.

A.9.2 Faking Integrity

As mentioned before, it is easy for *M* to overwrite the hashes and replace them with its own. It only needs to do the recalculations. What this variant makes difficult is detecting the check until the end of the connection, so we will discuss that here.

Suppose *M* sees the packet and wants to tell if a check is being used. Examination of the field holding the hash looks like random bits. The only other option is to try to reconstruct the chain. To do this, two things are needed:

1. the fields over which the hash chain is based
2. the salt

With the first packet, *M* has the first item. But it still needs the salt, which is not disclosed until the last packet in the chain.

From here, *M*'s only option is to try to hang on to a group of packets in an attempt to capture the salt from the last packet before it has to forward along the first packet. Depending on how long the chain is and how the connection is being used, this can deadlock or wedge the connection because responses (flow control updates, application messages, etc.) from *B* may be needed to elicit the rest of the chain from *A*.

A.9.3 Pros

- *M* cannot detect if a connection is HICCUPS-enabled until the end of the chain
- Can make an over-zealous *M* do more computation than you, thus raising the bar

A.9.4 Cons

- *M* can blast over the chain and insert its own, faking integrity
- Endpoint do not get integrity feedback until the end of the chain

A.9.5 Thoughts and Status

This variant stands apart from the rest in its ability to prevent detection of the check until the reveal is done.

We make the "raising the bar" argument by employing randomness in our protection strategy. The idea here is that we randomly protect some $1/N$ connections or $1/N$ packets. Since the middlebox cannot easily guess which packets are protected, it must overwrite hashes on all N of them if it wants a guarantee to fool us. This can be detected when we start seeing valid hash chains for connections and packets which were never protected in the first place.

Furthermore, since our solution is incrementally deployable, there may be connections that never run a check, and M will have to sort through those as well (although it could keep a history of hosts that never embed valid chains, but this is still extra work).

As we will show in the next two sections, this technique can also be combined with HashCash and AppSalt to make them stronger and further raise the bar on M.

A.10 Variant 9: HashCash with Reverse Hash Chain

This variant is a combination of HashCash and reverse hash chains. It requires the original hash of the chain be a special HashCash hash.

Throughout Connection:	Yes
Diagnostic Mode:	Yes, diagnostic only
	Mode Hidden? Yes, until chain revealed
Fields Used:	IPID
Raises Bar on M:	Yes, stronger than HashCash and reverse hash chains.

A.10.1 Detailed Description

Similar to the HashCash variant, A and B can precompute a pair (h,R) where R is a value that is added to the input of the hashing function. The value R causes the output h to have a property which is easily checked. Such a property could be that the hash begins with a minimum number of zeros.

In this variant, we now use the reverse hash cash to obscure the HashCash hash within a chain. The setup is the same as described for the reverse hash chain variant. The only differences are that the R value is used as the salt given by the `REVEAL`, and the original hash of the chain has the HashCash property.

The variant is outlined in Figure 16. As with HashCash, A brute forces through different salts to find a resulting h that meets the specified property. More specifically, the resultant hash h must be an element of the set of all hashes that meet the property, or $h \in P$.

The following order of events occurs for a length 4 chain:

1. A brute force searches for a value that yields h such that $h \in P$. This value is c_0.
2. A hashes (c_0, $fields_1$) to get c_1
3. A hashes c_1 to get c_2
4. A hashes c_2 to get c_3
5. A embeds c_3 into first packet and sends it
6. A embeds the hash of (c_2, $fields_2$) into the second packet
7. A embeds the hash of (c_1, $fields_3$) into the third packet

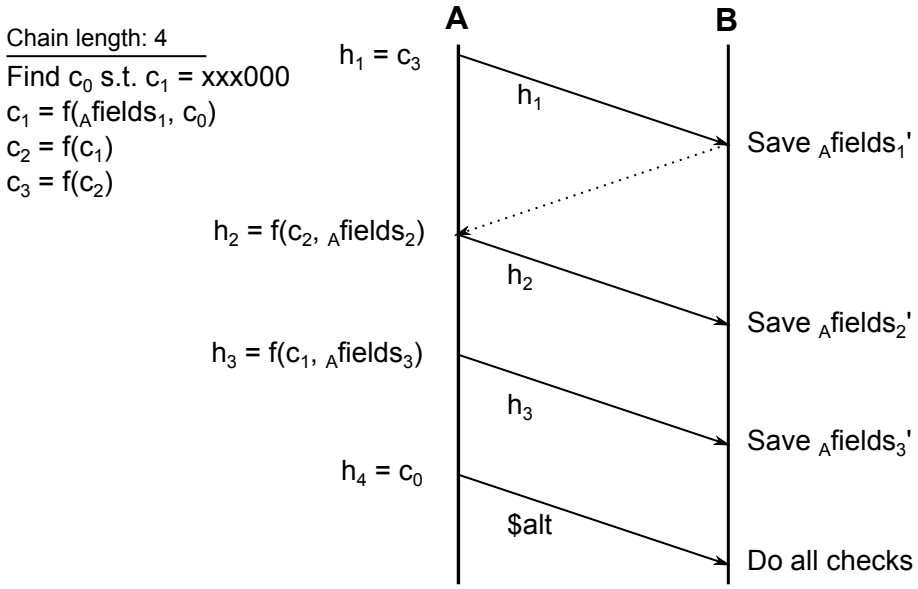

Figure 16: Timing diagram with no modifications

8. *A* embeds c_0 into the fourth and final packet

Basically it is exactly the same as the reverse hash chain variant, but we replace the random salt value with one that requires computational cycles. This makes it difficult for any middlebox to overwrite the chain since it must start with a value that is valid for the HashCash scheme.

A.10.2 Faking Integrity

In order for *M* to fool *A* and *B* into thinking that packets were not modified, it would have to overwrite the entire hash chain while ensuring that the HashCash property still holds. All by the time the full chain is sent, which should be very difficult. Also, if *M* adds a bunch of delay to the final packet of a chain, we should be able to detect that.

A.10.3 Pros

- A good combination of the strengths of HashCash (raising the bar computationally) and reverse hash chains (making mode detection difficult)

A.10.4 Cons

- Could only be used in a diagnostic connection

A.10.5 Thoughts and Status

This variant is worth continuing to explore. It gives all the same benefits of the reverse hash chain method with the added computational burden from the HashCashes. It can basically be viewed as an add-on for the reverse hash chains. Perhaps it could be used as a stronger assurance mode for more aggressively invasive middleboxes.

Coding the HashCash pool will be difficult if a CPU must precompute them in spare cycles. Without precomputation, it will probably be unusable for anything other than a diagnostic connection.

A.11 Variant 10: AppSalt

Uses application data in the integrity hashes to make them hard to modify without affecting the user experience.

Throughout Connection:	Yes
Diagnostic Mode:	No
Fields Used:	ISN, IPID
Raises Bar on *M*:	Yes. *M* would have to be a terminating proxy and cache lots of packets.

A.11.1 Detailed Description

This variant builds on the opportunistic approach from Section A.2 and protects the SYN integrity value with future *application-layer content* from a data packet *yet to be sent*. This ephemeral secret is difficult for a middlebox to reliably determine *a priori*. As before, the integrity value is encoded in the ISN of the SYN, but now the receiving end host, as well as any middleboxes, must know the contents of future application data in order to interpret the integrity.

For the ephemeral application-layer secret, the first data packet need not be a full MSS (e.g., in the case of an HTTP GET request). We therefore examined the initial application payload of each flow in a full day of border traffic from our organization. Among application data payloads of 6,742,466 flows, we find 5,377,440 (approximately 80 percent) where the first 40 bytes are unique. The 99th percentile of the distribution is that payloads appear twice, implying that 40 bytes of ephemeral secret is a reasonable lower-bound to prevent trivial guessing.

To illustrate the complete HICCUPS operation, we present a scenario where a web client connects to a server by performing the 3WHS and then issues an HTTP GET request for a specific resource. Neither the remote server nor any in-path middleboxes can reliably ascertain what will be the application data at the time the SYN is observed. Only the web client knows with certainty the initial HTTP application data that will be sent. In this example, the application layer data might contain such items as the GET URL, the host parameter, and the user agent string as shown in the example of Figure 17.

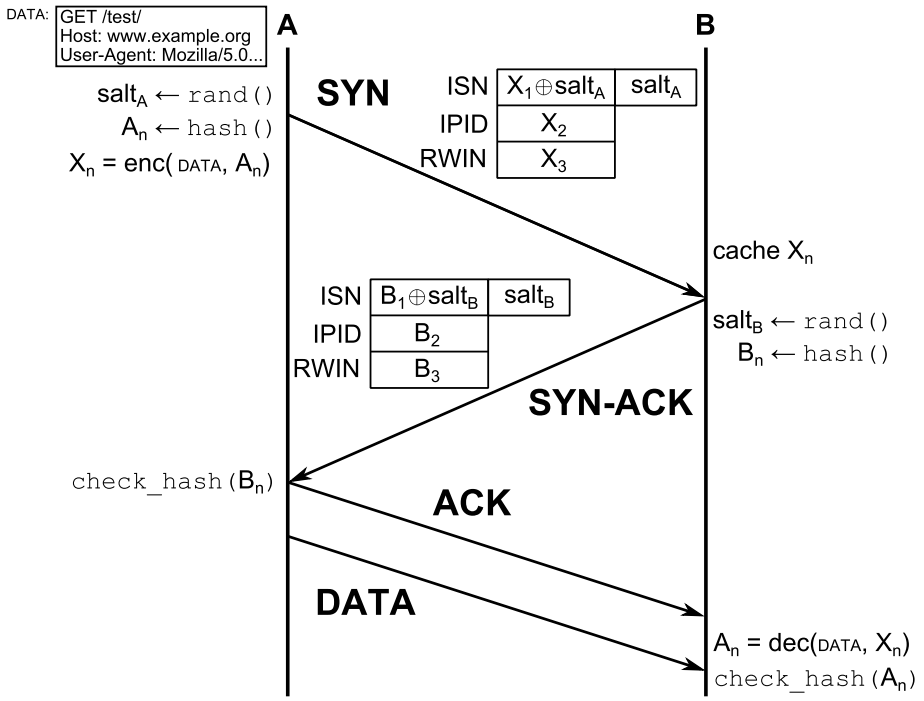

Figure 17: Timing diagram with no modifications

A.11.2 Faking Integrity

Since the application data needed to properly decode the SYN's integrity is not available to *M* at the time the SYN is received, it is difficult for *M* to check whether a connection is HICCUPS-enabled. Encoding integrity with future application data also increases the difficulty for a middlebox to tamper with a packet and evade detection. *M* cannot simply recalculate a new valid integrity. The ephemeral secret forces *M* to process the SYN packet before it can observe the application data. Otherwise, *M* has two remaining options to modify the packet headers and evade detection: make a best guess of the application data, or perform a man-in-the-middle (MITM) attack and fake a SYN-ACK response, inducing *A* to expose the application data secret.

M may attempt to guess the unseen application data (e.g., by using a profile of prior connections from *A* to *B*). However, *M* is unlikely to guess correctly for every connection between all pairs of hosts. If *M* guesses incorrectly, integrity values will not validate and the manipulations can be detected. Of course, *M* could change the actual application data to match its guess, but doing so fundamentally alters the application-layer behavior of the

connection.

In order to know the application data with certainty, *M* must act as a TCP-terminating proxy, a behavior that is detectable based on timing and by issuing connections to known unreachable hosts as shown in [12]. This MITM behavior, whereby *M* falsely claims to be *B*, spoofs the SYN-ACK and intercepts the resulting traffic, permits *M* to rebuild the original SYN with an updated integrity value and forward it along to the true destination. The non-spoofed SYN-ACK from *B* would have to be intercepted and the cached data from *A* could be sent. This situation is clearly more complicated than just the translating of sequence numbers; the middlebox has broken a connection and now has to marshal data between them, in addition to sending spoofed packets, buffering data, and rebuilding integrity values. Further, the middlebox must do this for all connections, potentially representing many endpoints.

A.11.3 Pros

- Very strong against middleboxes intending to perform undetected tampering
- Ties attempted evasion by a middlebox to the user experience

A.11.4 Cons

- Further blurs lines between layers. Forces us to understand application data at the TCP layer.
- Need to modify many applications to provide this data to the TCP stack at connection time

A.11.5 Thoughts and Status

We took the step of verifying that the system call to `connect()` initiates the 3WHS. The SYN is sent before any calls to `send()` are ever made.

Our current implementation of HICCUPS uses this variant as its protection scheme. We had to modify the kernel's socket API so that an application could specially request protection via AppSalt if it desired it. For more details, see Section 7.2.

References

[1] R. Craven *et al.*, "Techniques for the detection of faulty packet header modifications," Naval Postgraduate School, Tech. Rep. NPS-CS-14-002, Mar. 2014.

[2] R. Craven *et al.*, "A middlebox-cooperative TCP for a non end-to-end Internet," in *Proceedings of the 2014 ACM SIGCOMM Conference*, Chicago, IL, Aug. 2014.

[3] J. Postel. (1981, Sep.). Transmission control protocol. RFC 793 (Internet Standard). [Online]. Available: http://tools.ietf.org/html/rfc793

[4] D. Clark, "The design philosophy of the DARPA Internet protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 106–114, Aug. 1988.

[5] D. D. Clark *et al.*, "Tussle in cyberspace: Defining tomorrow's Internet," *IEEE/ACM Trans. Netw.*, vol. 13, no. 3, pp. 462–475, Jun. 2005.

[6] J. H. Saltzer *et al.*, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[7] H. Schulze and K. Mochalski, "Internet study 2008/2009," ipoque GmbH, Leipzig, Germany, Tech. Rep., 2009.

[8] M. Allman *et al.* (2009, Sep.). TCP congestion control. RFC 5681 (Draft Standard). [Online]. Available: http://tools.ietf.org/html/rfc5681

[9] R. Braden *et al.* (1988, Sep.). Computing the Internet checksum. RFC 1071 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc1071

[10] B. Carpenter and S. Brim. (2002, Feb.). Middleboxes: Taxonomy and issues. RFC 3234 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc3234

[11] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proceedings of the 2012 ACM SIGCOMM Conference*, Aug. 2012, pp. 13–24.

[12] C. Kreibich *et al.*, "Netalyzr: Illuminating the edge network," in *Proceedings of the 2010 ACM SIGCOMM Internet Measurement Conference*, Melbourne, Australia, Nov. 2010, pp. 246–259.

[13] Cisco Systems. (2011, Oct.). Single TCP flow performance on firewall services module (FWSM). [Online]. Available: https://supportforums.cisco.com/docs/DOC-12668#TCP_Sequence_Number_Randomization_and_SACK

[14] M. Smart *et al.*, "Defeating TCP/IP stack fingerprinting," in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.

[15] G. Fisk *et al.*, "Eliminating steganography in Internet traffic with active wardens," in *Revised Papers from the 5th International Workshop on Information Hiding*, Noordwijkerhout, The Netherlands, Oct. 2002, pp. 18–35.

[16] C. B. Smith and S. S. Agaian, "Denoising and the active warden," in *IEEE International Conference on Systems, Man and Cybernetics*, Montreal, Canada, Oct. 2007, pp. 3317–3322.

[17] M. Handley *et al.*, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics," in *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, Aug. 2001.

[18] R. Fonseca *et al.*, "IP options are not an option," EECS UC Berkeley, Tech. Rep. 2005-24, Dec. 2005.

[19] M. Honda *et al.*, "Is it still possible to extend TCP?" in *Proceedings of the 2011 ACM SIGCOMM Internet Measurement Conference*, Berlin, Germany, Nov. 2011, pp. 181–194.

[20] S. Bauer *et al.*, "Measuring the state of ECN readiness in servers, clients, and routers," in *Proceedings of the 2011 ACM SIGCOMM Internet Measurement Conference*, Berlin, Germany, Nov. 2011, pp. 171–180.

[21] A. Ford *et al.* (2013, Jan.). TCP extensions for multipath operation with multiple addresses. RFC 6824 (Experimental). [Online]. Available: http://tools.ietf.org/html/rfc6824

[22] V. Sekar *et al.*, "The middlebox manifesto: Enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, Cambridge, MA, Nov. 2011.

[23] S. Radhakrishnan *et al.*, "TCP fast open," in *Proceedings of the 7th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Tokyo, Japan, Dec. 2011.

[24] A. Bittau *et al.*, "The case for ubiquitous transport-level encryption," in *Proceedings of the 19th USENIX Conference on Security*, Aug. 2010, p. 26.

[25] M. Belshe, "Improving web protocols: SPDY performance data," Apr. 2011, presentation at the University of Delaware. [Online]. Available: http://www.slideshare.net/mbelshe/university-of-delaware-improving-web-protocols-early-spdy-talk

[26] T. Wu, “Network neutrality, broadband discrimination,” *Journal on Telecommunications and High Technology Law*, vol. 2, p. 141, 2003.

[27] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison Wesley Longman, Inc., 1994.

[28] H. Zimmermann, “OSI reference model–the ISO model of architecture for open systems interconnection,” *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, Apr. 1980.

[29] J. Postel. (1980, Aug.). User datagram protocol. RFC 768 (Internet Standard). [Online]. Available: http://tools.ietf.org/html/rfc768

[30] R. Fielding *et al.* (1999, Jun.). Hypertext transfer protocol. RFC 2616 (Draft Standard). [Online]. Available: http://tools.ietf.org/html/rfc2616

[31] J. Postel. (1981, Sep.). Internet protocol. RFC 791 (Internet Standard). [Online]. Available: http://tools.ietf.org/html/rfc791

[32] K. Nichols *et al.* (1998, Dec.). Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. RFC 2474 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc2474

[33] K. Ramakrishnan *et al.* (2001, Sep.). The addition of explicit congestion notification (ECN) to IP. RFC 3168 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc3168

[34] B. Briscoe. (2014, Feb.). Reusing the IPv4 identification field in atomic packets. Internet draft. IETF Internet Area Working Group (intarea). [Online]. Available: http://tools.ietf.org/html/draft-briscoe-intarea-ipv4-id-reuse

[35] A. Ramaiah. (2012, Mar.). TCP option space extension. Internet draft. [Online]. Available: https://tools.ietf.org/html/draft-ananth-tcpm-tcpoptext

[36] J. Postel. (1981, Sep.). Internet control message protocol. RFC 792 (Internet Standard). [Online]. Available: http://tools.ietf.org/html/rfc792

[37] V. Jacobson. (1987). traceroute. [Online]. Available: ftp://ftp.ee.lbl.gov/traceroute.tar.gz

[38] B. Augustin *et al.*, “Avoiding traceroute anomalies with Paris traceroute,” in *Proceedings of the 2006 ACM SIGCOMM Internet Measurement Conference*, Rio de Janeriro, Brazil, Oct. 2006, pp. 153–158.

[39] G. Detal *et al.*, “Revealing middlebox interference with Tracebox,” in *Proceedings of the 2013 ACM SIGCOMM Internet Measurement Conference*, Barcelona, Spain, Oct. 2013, pp. 1–8.

[40] C. A. Kent and J. C. Mogul, “Fragmentation considered harmful,” *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 75–87, Jan. 1995.

[41] J. Heffner *et al.* (2007, Jul.). IPv4 reassembly errors at high data rates. RFC 4963 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc4963

[42] A. Medina *et al.*, “Measuring interactions between transport protocols and middleboxes,” in *Proceedings of the 2004 ACM SIGCOMM Internet Measurement Conference*, Taormina, Italy, Oct. 2004, pp. 336–341.

[43] A. Medina *et al.*, “Measuring the evolution of transport protocols in the Internet,” *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.

[44] M. Luckie and B. Stasiewicz, “Measuring path MTU discovery behaviour,” in *Proceedings of the 2010 ACM SIGCOMM Internet Measurement Conference*, Melbourne, Australia, Nov. 2010, pp. 102–108.

[45] K. Lahey. (2000, Sep.). TCP problems with path MTU discovery. RFC 2923 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc2923

[46] M. Mathis and J. Heffner. (2007, Mar.). Packetization layer path MTU discovery. RFC 4821 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc4821

[47] J. Semke *et al.*, “Automatic TCP buffer tuning,” *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 315–323, Oct. 1998.

[48] N. Spring *et al.* (2003, Jun.). Robust explicit congestion notification (ECN) signaling with nonces. RFC 3540 (Experimental). [Online]. Available: http://tools.ietf.org/html/rfc3540

[49] J. Postel. (1983, Nov.). The TCP maximum segment size and related topics. RFC 879. [Online]. Available: http://tools.ietf.org/html/rfc879

[50] D. Borman. (2012, Jul.). TCP options and maximum segment size (MSS). RFC 6691 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc6691

[51] M. Mathis *et al.* (1996, Oct.). TCP selective acknowledgment options. RFC 2018 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc2018

[52] V. Jacobson *et al.* (1992, May). TCP extensions for high performance. RFC 1323 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc1323

[53] B. Hesmans *et al.*, "Are TCP extensions middlebox-proof?" in *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes*, Santa Barbara, CA, Dec. 2013, pp. 37–42.

[54] S. Karandikar *et al.*, "TCP rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 45–58, Jan. 2000.

[55] M. Barbera *et al.*, "Active window management: An efficient gateway mechanism for TCP traffic control," in *IEEE International Conference on Communications*, Glasgow, Scotland, Jun. 2007, pp. 6141–6148.

[56] M. Dischinger *et al.*, "Glasnost: Enabling end users to detect traffic differentiation," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, Apr. 2010, pp. 405–418.

[57] corbet. (2004, Jul.). TCP window scaling and broken routers. [Online]. Available: https://lwn.net/Articles/92727/

[58] Anonymous, "Private communication," 2011.

[59] T. Flach *et al.*, "Reducing web latency: The virtue of gentle aggression," in *Proceedings of the 2013 ACM SIGCOMM Conference*, Hong Kong, China, Aug. 2013, pp. 159–170.

[60] M. Honda *et al.*, "Rekindling network protocol innovation with user-level stacks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 52–58, Apr. 2014.

[61] J. Daemen and V. Rijmen, *The Design of Rijndael*. Secaucus, NJ: Springer-Verlag New York, Inc., 2002.

[62] D. Eastlake and T. Hansen. (2011, May). US secure hash algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc6234

[63] R. Beverly *et al.*, "Understanding the efficacy of deployed Internet source address validation filtering," in *Proceedings of the 2009 ACM SIGCOMM Internet Measurement Conference*, Chicago, IL, Nov. 2009, pp. 356–369.

[64] S. Kent and K. Seo. (2005, Dec.). Security architecture for the Internet protocol. RFC 4301 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc4301

[65] S. Kent. (2005, Dec.). IP encapsulating security payload (ESP). RFC 4303 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc4303

[66] S. Kent. (2005, Dec.). IP authentication header. RFC 4302 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc4302

[67] B. Aboba and W. Dixon. (2004, Mar.). IPsec-network address translation (NAT) compatibility requirements. RFC 3715 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc3715

[68] A. Heffernan. (1998, Aug.). Protection of BGP sessions via the TCP MD5 signature option. RFC 2385 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc2385

[69] R. Rivest. (1992, Apr.). The MD5 message-digest algorithm. RFC 1321 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc1321

[70] CERT. (1995, Jan.). IP spoofing attacks and hijacked terminal connections. Advisory number CA-1995-01. [Online]. Available: https://www.cert.org/historical/advisories/CA-1995-01.cfm

[71] CERT. (2004, Apr.). The border gateway protocol relies on persistent TCP sessions without specifying authentication requirements. Vulnerability Note VU#415294. [Online]. Available: http://www.kb.cert.org/vuls/id/415294

[72] J. Touch *et al.* (2010, Jun.). The TCP authentication option. RFC 5925 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc5925

[73] G. Lebovitz and E. Rescorla. (2010, Jun.). Cryptographic algorithms for the TCP authentication option. RFC 5926 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc5926

[74] A. Freier *et al.* (2011, Aug.). The secure sockets layer (SSL) protocol version 3.0. RFC 6101 (Historic). [Online]. Available: http://tools.ietf.org/html/rfc6101

[75] S. Birkner. (2006). Man-in-the-middle attack of Diffie-Hellman key agreement.svg. Creative Commons Attribution-Share Alike 3.0. [Online]. Available: https://commons.wikimedia.org/wiki/File: Man-in-the-middle_attack_of_Diffie-Hellman_key_agreement.svg

[76] N. Williams and M. Richardson. (2008, Nov.). Better-than-nothing security: An unauthenticated mode of IPsec. RFC 5386 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc5386

[77] C. Shue *et al.*, "Analysis of IPsec overheads for VPN servers," in *Proceedings of the First Workshop on Secure Network Protocols*, Boston, MA, Nov. 2005, pp. 25–30.

[78] A. Langley, "Opportunistic encryption everywhere," in *Web 2.0 Security and Privacy (W2SP)*, Oakland, CA, May 2009.

[79] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: A field study of middlebox failures in datacenters," in *Proceedings of the 2013 ACM SIGCOMM Internet Measurement Conference*, Barcelona, Spain, Oct. 2013, pp. 9–22.

[80] ABI Research. (2011, Jan.). Enterprise network and data security spending shows remarkable resilience. [Online]. Available: http://goo.gl/E5Unmb

[81] M. Walfish *et al.*, "Middleboxes no longer considered harmful," in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, San Francisco, CA, Dec. 2004.

[82] A. Knutsen *et al.* (2013, Feb.). TCP option for transparent middlebox negotiation. Internet draft. IETF Transport Area Working Group (tsvwg). [Online]. Available: http://tools.ietf.org/html/draft-ananth-middisc-tcpopt

[83] A. Gember *et al.*, "Toward software-defined middlebox networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, Redmond, WA, Oct. 2012.

[84] "Software-defined networking: The new norm for networks," White Paper, Open Networking Foundation, Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[85] J. W. Anderson *et al.*, "xOMB: Extensible open middleboxes with commodity servers," in *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Oct. 2012, pp. 49–60.

[86] G. Gibb *et al.*, "Outsourcing network functionality," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*, Aug. 2012, pp. 73–78.

[87] V. Sekar *et al.*, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, Apr., 2012.

[88] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proceedings of the 2013 ACM SIGCOMM Conference*, Aug. 2013, pp. 27–38.

[89] E. Kohler *et al.*, "The Click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[90] Hurricane Electric Internet Services. (2014, May). Internet statistics. [Online]. Available: http://bgp.he.net/report/netstats

[91] *Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. 802.3-2008, 2008.

[92] *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std. 802.11-2012, 2012.

[93] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 309–319, Aug. 2000.

[94] P. Eckersley, "Switzerland design," Electronic Frontier Foundation, San Francisco, CA, May 2008. [Online]. Available: http://sourceforge.net/p/switzerland/code/HEAD/tree/trunk/doc/design.pdf

[95] L. MartinGarcia. (2011, Apr.). Nping echo protocol. [Online]. Available: https://svn.nmap.org/nmap/nping/docs/EchoProtoRFC.txt

[96] S. Sanfilippo. (2005). hping. [Online]. Available: http://www.hping.org/

[97] M. Luckie *et al.*, "Traceroute probe method and forward IP path inference," in *Proceedings of the 2008 ACM SIGCOMM Internet Measurement Conference*, Oct. 2008, pp. 311–324.

[98] F. Baker. (1995, Jun.). Requirements for IP version 4 routers. RFC 1812 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc1812

[99] D. Malone and M. Luckie, "Analysis of ICMP quotations," in *Passive and Active Network Measurement*, ser. Lecture Notes in Computer Science, S. Uhlig *et al.*, Eds. Springer Berlin Heidelberg, 2007, vol. 4427, pp. 228–232.

[100] CERT. (2003, Jun.). Linux kernel IP stack incorrectly calculates size of an ICMP citation for ICMP errors. Vulnerability Note VU#471084. [Online]. Available: http://www.kb.cert.org/vuls/id/471084

[101] W. Albert and T. Tullis, *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics*, 2nd ed. Waltham, MA: Elsevier Inc., 2013, p. 203.

[102] N. Weaver *et al.*, "Here be web proxies," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, M. Faloutsos and A. Kuzmanovic, Eds. Springer International Publishing, 2014, vol. 8362, pp. 183–192.

[103] J. Touch. (2013, Feb.). Updated specification of the IPv4 ID field. RFC 6864 (Proposed Standard). [Online]. Available: http://tools.ietf.org/html/rfc6864

[104] The Cooperative Association for Internet Data Analysis (CAIDA). (2012, Apr.). Passive monitor: equinix-sanjose. [Online]. Available: http://www.caida.org/data/monitors/passive-equinix-sanjose.xml

[105] W. Eddy. (2007, Aug.). TCP SYN flooding attacks and common mitigations. RFC 4987 (Informational). [Online]. Available: http://tools.ietf.org/html/rfc4987

[106] P. McManus. (2008, Apr.). Improving syncookies. [Online]. Available: http://lwn.net/Articles/277146/

[107] E. Cole, *Hiding in Plain Sight: Steganography and the Art of Covert Communication*. Indianapolis, IN: Wiley Publishing Inc., 2003.

[108] N. R. Bennett, "JPEG steganalysis and TCP/IP steganography," M.S., University of Rhode Island, 2009.

[109] X. Luo *et al.*, "CLACK: A network covert channel based on partial acknowledgment encoding," in *IEEE International Conference on Communications*, Dresden, Germany, Jun. 2009, pp. 1–5.

[110] M. Allman, "Comments on selecting ephemeral ports," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 2, pp. 13–19, Mar. 2009.

[111] A. Appleby. (2011). Murmurhash 3.0. [Online]. Available: https://sites.google.com/site/murmurhash/

[112] L. Torvalds. (2013, May). Linux kernel v3.9.4. [Online]. Available: http://www.kernel.org

[113] R. Craven *et al.* (2014, Jun.). TCP HICCUPS Linux 3.9.4 kernel patch. [Online]. Available: http://hdl.handle.net/10945/41959

[114] R. Craven *et al.* (2014). CMAND: TCP HICCUPS. [Online]. Available: http://tcphiccups.org

[115] Google, Inc. (2013). chromium code search. [Online]. Available: https://code.google.com/p/chromium/codesearch#chromium/src/net/socket/tcp_client_socket_libevent.cc&sq=package:chromium&type=cs&l=312

[116] S. Rostedt. (2008). ftrace - function tracer. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[117] P. Chifflier. (2012). nfqueue-bindings. [Online]. Available: https://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki/

[118] P. Biondi. (2012). Scapy. [Online]. Available: http://www.secdev.org/projects/scapy/

[119] B. Chun *et al.*, "PlanetLab: an overlay testbed for broad-coverage services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003.

[120] Y. Hyun and k. claffy. (2014). Archipelago (Ark) measurement infrastructure. [Online]. Available: http://www.caida.org/projects/ark/

[121] A.-J. Su *et al.*, "Drafting behind Akamai (Travelocity-based detouring)," in *Proceedings of the 2006 ACM SIGCOMM Conference*, Sep. 2006, pp. 435–446.

[122] Z. Wang *et al.*, "An untold story of middleboxes in cellular networks," in *Proceedings of the 2011 ACM SIGCOMM conference*, Toronto, Canada, Aug. 2011, pp. 374–385.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California